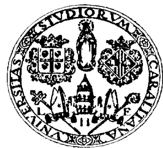

Università degli Studi di Cagliari

DOTTORATO DI RICERCA

INGEGNERIA ELETTRONICA ED INFORMATICA

Ciclo XXV

Integrated support for Adaptivity and Fault-tolerance in MPSoCs

Settore scientifico disciplinare di afferenza

ING-INF/01 ELETTRONICA

Presentata da: GIUSEPPE TUVERI

Coordinatore Dottorato PROF. ALESSANDRO GIUA

Tutor/Relatore PROF. LUIGI RAFFO

Esame finale anno accademico 2011 – 2012

Ph.D. in Electronic and Computer Engineering
Dept. of Electrical and Electronic Engineering
University of Cagliari

Integrated support for Adaptivity and Fault-tolerance in MPSoCs

Giuseppe TUVERI

Advisor: Prof. Luigi RAFFO
Curriculum: ING-INF/01 Elettronica

XXV Cycle
April 2013

Ph.D. in Electronic and Computer Engineering
Dept. of Electrical and Electronic Engineering
University of Cagliari

Integrated support for Adaptivity and Fault-tolerance in MPSoCs

Giuseppe TUVERI

Advisor: Prof. Luigi RAFFO
Curriculum: ING-INF/01 Elettronica

XXV Cycle
April 2013

EBBÉ ECCO

Abstract

The technology improvement and the adoption of more and more complex applications in consumer electronics are forcing a rapid increase in the complexity of multiprocessor systems on chip (MPSoCs). Following this trend, MPSoCs are becoming increasingly dynamic and adaptive, for several reasons. One of these is that applications are getting intrinsically dynamic. Another reason is that the workload on emerging MPSoCs cannot be predicted because modern systems are open to new incoming applications at run-time. A third reason which calls for adaptivity is the decreasing component reliability associated with technology scaling. Components below the 32-nm node are more inclined to temporal or even permanent faults. In case of a malfunctioning system component, the rest of the system is supposed to take over its tasks. Thus, the system adaptivity goal shall influence several design decisions, that have been listed below: 1) The applications should be specified such that system adaptivity can be easily supported. To this end, we consider Polyhedral Process Networks (PPNs) as model of computation to specify applications. PPNs are composed by concurrent and autonomous processes that communicate between each other using bounded FIFO channels. Moreover, in PPNs the control is completely distributed, as well as the memories. This represents a good match with the emerging MPSoC architectures, in which processing elements and memories are usually distributed. Most importantly, the simple operational semantics of PPNs allows for an easy adoption of system adaptivity mechanisms. 2) The hardware platform should guarantee the flexibility that adaptivity mechanisms require. Networks-on-Chip (NoCs) are emerging communication infrastructures for MPSoCs that, among many other advantages, allow for system adaptivity. This is because NoCs are generic, since the same platform can be used to run different applications, or to run the same application with different mapping of processes. However, there is a mismatch between the generic structure of the NoCs and the semantics of the PPN model. Therefore, in this thesis we investigate and propose several communication approaches to overcome this mismatch. 3) The system must be able to change the process mapping at run-time, using process migration. To this end, a process migration mechanism has been proposed and evaluated. This mechanism takes into account specific requirements of the embedded domain such as predictability and efficiency. To face the problem of graceful degradation of the system, we enriched the MADNESS NoC platform by adding fault tolerance support at both software and hardware level. The proposed process migration mechanism can be exploited to cope with permanent faults by migrating the processes running on the faulty processing element. A fast heuristic is used to determine the new mapping of the processes to tiles. The experimental results prove that the overhead in terms of execution time, due to the execution time of the remapping heuristic, together with the actual process migration, is almost negligible

compared to the execution time of the whole application. This means that the proposed approach allows the system to change its performance metrics and to react to faults without a substantial impact on the user experience.

Contents

1 Introduction **1**
1.1 Main goals and thesis organization 1

2 State of the art **3**
2.1 System adaptivity for MPSoCs 3
2.1.1 Open issues and related works 4
2.2 Fault-tolerance in NoC-based MPSoCs 6
2.2.1 Open issues and related works 6

3 The MADNESS Project **9**
3.1 Framework outline 9
3.2 High-level DSE 11
3.3 FPGA-based evaluation environment 13
3.4 Compilation toolchain and hardware abstraction layer 14
3.4.1 Compilation toolchain 14
3.4.2 Hardware abstraction layer 14
3.5 Support for Adaptivity 15
3.6 Support for Fault-tolerance 15

4 The MADNESS evaluation platform **17**
4.1 Platform overview 17
4.2 System-level platform description input file 18
4.3 The soft IP cores RTL repository 21
4.3.1 Elements of the Open Core communication Protocol (OCP) 21
4.3.2 Processing elements, memories and I/O 22
4.3.3 Interconnection elements 23
4.3.4 Message Passing programming model 25
4.3.5 Message Passing hardware support 25
4.3.6 Interrupt generation support 26
4.3.7 Software libraries 26
4.3.8 Shared memory support: synchronization modules 28
4.4 The SHMPI platform builder 29
4.5 Performance extraction 30

5 Methodologies for adaptive MPSoCs **33**

5.1 PPN-over-NoC communication . . . 34
5.1.1 Some definitions . . . 34
5.1.2 Inter-tile synchronization problem . . . 35
5.1.3 Virtual connector approach (VC) . . . 37
5.1.4 Virtual connector with variable rate approach (VRVC) . . . 38
5.1.5 Request-driven approach (R) . . . 39
5.2 Process migration . . . 40
5.2.1 Migratable PPN process structure . . . 41
5.2.2 Process migration mechanism . . . 43
5.3 Experiments and results . . . 45
5.3.1 Case studies and MPSoC platform setup . . . 45
5.3.2 Inter-tile communication efficiency . . . 47
5.3.3 Process migration benefits and overhead . . . 50

6 Fault-tolerance support within the MADNESS framework 55
6.1 Proposed approach . . . 55
6.1.1 Fault detection . . . 55
6.1.2 Task migration hardware module . . . 56
6.1.3 Online task remapping strategies . . . 58
6.2 Experiments and results . . . 58
6.2.1 Case studies . . . 59
6.2.2 Flow control functionality assessment . . . 60
6.2.3 Remapping heuristic and process migration execution time overhead . 61
6.2.4 Evaluation of the remapping strategy . . . 62
6.2.5 Architectural support hardware overhead . . . 64

7 Conclusions and future developments 69

Bibliography 71

List of Figures

3.1 The MADNESS framework . . . 10
3.2 Scenario-based DSE. . . . 12

4.1 Overview of the evaluation platform . . . 18
4.2 ×pipes switch architecture . . . 23
4.3 A general overview of an example template instance . . . 25

5.1 Software infrastructure for each tile of the NoC. . . . 34
5.2 Producer-consumer pair with FIFO buffer split over two tiles. . . . 35
5.3 Example of a PPN and structure of process $P2$ . . . 35
5.4 Producer-consumer pair using the virtual connector approach. . . . 36
5.5 Pseudocode of the VC approach. . . . 37
5.6 Producer-consumer implementation . . . 38
5.7 Pseudocode of the R approach. . . . 39
5.8 Migration diagram. . . . 41
5.9 Migratable PPN process. . . . 42
5.10 PPN specification of the Sobel filter. . . . 45
5.11 PPN specification of the M-JPEG encoder. . . . 46
5.12 Structure of middleware- and network- level packets. . . . 47
5.13 Fixed mappings for Sobel and M-JPEG . . . 48
5.14 Total execution time for different communication approaches. . . . 49
5.15 Slowdown for different communication approaches. . . . 49
5.16 Traffic injected into the NoC by executing Sobel . . . 49
5.17 M-JPEG process scheduling when running on a single tile. . . . 51
5.18 M-JPEG process scheduling while migrating P_2 . . . 51
5.19 Execution time and generated traffic as a function of the process mapping . . . 53

6.1 Interface and internal block diagram of the task migration hardware module . . . 57
6.2 PPN specification of the M-JPEG encoder. . . . 59
6.3 Simplified PPN specification of the H.264 decoder. . . . 60
6.4 Impact of the interrupt-based request messages on the RD flow control . . . 60
6.5 M-JPEG process scheduling when migrating M_1 . . . 61
6.6 Initial mapping and the two single-fault scenarios . . . 63
6.7 Comparison of performance degradation when n_1 is faulty . . . 63
6.8 Comparison of performance degradation when n_2 is faulty . . . 63
6.9 Initial mapping and the two single-fault scenarios . . . 64

6.10 Comparison of performance degradation when n_1 is faulty 64
6.11 Comparison of performance degradation when n_2 is faulty 65
6.12 Area occupation overhead in comparison to the baseline network adapter 65
6.13 Critical path length overhead 66
6.14 Area occupation overhead 67
6.15 Area overhead dependence on the supported number of channels 67

List of Tables

4.1 Implemented OCP signals 22

5.1 Middleware table example 40
5.2 Execution times of Sobel functions 45
5.3 Execution times of M-JPEG functions 46

6.1 Execution times of M-JPEG processes 59
6.2 Execution times of H.264 processes 60
6.3 Calculation times of remapping heuristics 62

Chapter 1

Introduction

1.1 Main goals and thesis organization

Modern embedded systems become increasingly dynamic and adaptive. This is true for both application and architecture. Regarding application, we can distinguish between two classes of dynamic behaviour: intra-application and inter-application. An example of intra-application dynamic behaviour is when an application aims at keeping an estabilished level of Quality of Service (QoS) under varying circumstances. For example, a video application could dynamically lower its resolution to decrease its computational demands in order to reduce the battery drain. Inter-application adaptive behaviour is caused by the fact that modern embedded systems often require to support an increasing amount of applications and standards. Also, where such systems used to be "closed systems" in the past, today's embedded systems become more and more "open systems" for which third-party software applications can be downloaded and installed, or updated. As a consequence, the application workload in such systems (i.e., the applications that are concurrently executing and contending for system resources), and therefore the intensity and nature of the application demands, can be very fluctuating. For this reason, the notion of workload scenarios has gained considerable popularity in the past years [36], [16]. The underlying computer architectures of embedded systems also become more and more adaptive and dynamic. For example, reconfigurable hardware components such as Field Programmable Gate Arrays (FPGAs) have become popular architectural elements that allow for accelerating specific computational kernels in applications (e.g., [44]). Moreover, there has been a growing interest on reconfigurable on-chip networks (e.g., [20]). Such reconfigurable networks allow for adapting the network (in terms of e.g. QoS requirements, topology, etc.) to the demands of the application(s) that are being executed. Improving system reliability is another area in which the notion of adaptivity is gaining research attention. In the case of, e.g., a malfunctioning architecture component, other components in the system could take over the task(s) of the faulty component. This either requires component duplication and some voting mechanism (which may be undesirable for many embedded systems in which system costs play a key role) or adaptive system behaviour that allows for a redistribution of application tasks to architecture resources.
The above trend towards adaptivity and dynamic systems causes an important anomaly in design methods for modern embedded systems. Traditionally, the mapping of applications

onto the underlying architectural components of an embedded system has always been done in a static fashion. Here, we refer to the term "mapping" as the process of allocating architectural resources that will be used by the application(s), and the spatial binding as well as scheduling of application tasks to these resources. This mapping process is of primary importance to embedded systems as it should carefully trade-off non-functional requirements (performance, power, costs, reliability, etc.). However, with the increasing adaptive and dynamic behaviour of systems, such mapping decisions cannot be made at design time anymore. A run-time system will be needed to dynamically map and re-map applications onto the (available) underlying architectural resources. Moreover, with deep-sub-micron technology, the possibility of experiencing faults in the circuitry is significant, requiring the system to feature support for graceful degradation of the performance in case of malfunctioning.
To cope with these issues, we have devised techniques that allow to change the mapping of the application processes onto the processing cores at run-time. The development of these techniques required the introduction of dedicated support at several levels. At the architectural level, we considered a distributed-memory tile-based template, where tiles are interconnected through a NoC, to support the high flexibility and scalability demands. The architectural template is customizable in terms of the number of processors and network topology. It has been extended with newly developed hardware IPs that facilitate the run-time management and that expose to the applications the needed communication and synchronization primitives, referring to a message-passing model of computation. Extensions will be described in Chapter 4. At the software level, a specific layered infrastructure has been devised, that is actually in charge of managing the mapping of the tasks, the communication between them and the migration process. The software/middleware infrastructure will be described in Chapter 5. Eventually, fault-tolerance support has been introduced at both software and hardware levels. The idea is to exploit the migration method in case of run-time faults in the processing cores. The tasks mapped on faulty cores have to be migrated to fault-free ones at run-time, so that the application can continue its execution without disruption. To this aim, several extensions to the process migration mechanism are required. The details of the proposed techniques for fault-tolerance will be described in Chapter 6.

Chapter 2

State of the art

2.1 System adaptivity for MPSoCs

The technology improvement and the adoption of more and more complex applications in consumer electronics are forcing a rapid increase in the complexity of multiprocessor systems on chip (MPSoCs). Following this trend, MPSoCs are becoming increasingly dynamic and adaptive, for several reasons. One of these is that applications are getting intrinsically dynamic. Another reason is that the workload on emerging MPSoCs cannot be predicted because modern systems are open to new incoming applications at run-time. A third reason which calls for adaptivity is the decreasing component reliability associated with technology scaling. Components below the 32-nm node are more inclined to temporal or even permanent faults. In case of a malfunctioning system component, the rest of the system is supposed to take over its tasks.

Thus, the system adaptivity goal shall influence several design decisions, that have been listed below:

1) The applications should be specified such that system adaptivity can be easily supported. To this end, we consider Polyhedral Process Networks (PPNs) [45], a special class of Kahn Process Networks (KPNs) [22], as model of computation to specify applications. PPNs are composed by concurrent and autonomous processes that communicate between each other using bounded FIFO channels. Moreover, in PPNs the control is completely distributed, as well as the memories. This represents a good match with the emerging MPSoC architectures, in which processing elements and memories are usually distributed. Most importantly, the simple operational semantics of PPNs allows for an easy adoption of system adaptivity mechanisms. For instance, the process state which has to be transferred upon process migration does not have to be specified by hand by the designer and can be smaller compared to other solutions.

2) As a second design decision, the hardware platform should guarantee the flexibility that adaptivity mechanisms require. Networks-on-Chip (NoCs) [12], which is the platform model considered within this thesis, are emerging communication infrastructures for MPSoCs that, among many other advantages, allow for system adaptivity. This is because NoCs are generic, since the same platform can be used to run different applications, or to run the same application with different mapping of processes. However, there is a mismatch between the generic structure of the NoCs and the semantics of the PPN model of compu-

tation (MoC). Therefore, in this thesis we investigate and propose several communication approaches to overcome this mismatch.

3) Finally, the system must be able to change the process mapping at run-time, using process migration. To this end, a process migration mechanism has been proposed and evaluated. This mechanism takes into account specific requirements of the embedded domain such as predictability and efficiency. The efficiency of the proposed process migration mechanism depends on the design decisions discussed above, such as the MoC used to specify the applications. In this respect, the adoption of the PPN MoC ease the realization of process migration, when using the proposed approach. During our research activities, we found that the problem of a predictable and efficient process migration mechanism in distributed-memory MPSoCs has not received sufficient attention. The aim of the work done over these years is to contribute to a more mature solution of this problem.

2.1.1 Open issues and related works

Run-time resource management is a known topic in general purpose distributed systems scheduling [9]. In particular, process migration mechanisms [40, 29], have been developed and evaluated in this context to enable dynamic load distribution, fault resilience, and improved system administration and data access locality. In recent years, run-time management is gaining popularity and applications also in multiprocessor embedded systems. This domain imposes tight constraints, such as cost, power, and predictability, that run-time management and process migration mechanisms must consider carefully. [34] provides a survey of run-time management examples in state-of-the-art academic and industrial solutions, together with a generic description of run-time manager features and design space.

A relevant part of the work spent working on adaptivity topics has been focused on a specific component of run-time management strategies, namely the process migration mechanism. Papers addressing process (or task) migration implementation in MPSoCs can also be found in the literature. The closest to the presented work is [4], in which the goals of scalability and system adaptivity are achieved through a distributed task migration decision policy over a purely distributed-memory multiprocessor. Similar to our approach, their platform is programmed using a process network MoC. However, in their approach the actual task migration can take place only at fixed points, which correspond to the communication primitive calls. The described approach, instead, enables migration at any point in the execution of the main body of processes. This leads to a faster response time to migration decisions, which is preferable for instance in case of faults.

Other task migration approaches are explained and quantitatively evaluated in [7] and [3]. Dynamic task re-mapping is achieved at user-level or middleware/OS level respectively. In both these approaches, the user needs to define checkpoints in the code where the migration can take place. This can require some manual effort from the designer which is not needed in the proposed approach. Moreover, a relevant difference from the presented approach is the inter-task communication realization, which exploits a shared memory system. We argue that our approach, which uses purely distributed memory, can perform better in emerging MPSoC platforms since it provides better scalability.

The model of computation that we have adopted (Polyhedral Process Networks [45]) not only eases significantly the implementation of system adaptivity mechanism, but it also has several other advantages and applications which can be found in the literature. In particular, the proposed approach exploits the pn compiler [46] to automatically convert static affine

nested loop programs (SANLPs) to parallel PPN specifications and to determine the buffer sizes that guarantee deadlock-free execution. Thus, usage of the PPN model of computation allows us to program an MPSoC in a systematic and automated way. Although the pn compiler imposes some restrictions on the specification of the input application, we note that a large set of streaming applications can be effectively specified as SANLPs. In addition to the case studies considered in this thesis, more application examples regard image/video processing (JPEG2000, H.264), sound processing (FM radio, MP3), and scientific computation (QR decomposition, stencil, finite-difference time-domain). Moreover, a recent work [41] has shown that most of the streaming applications can be specified using the Synchronous Data Flow (SDF) model [25]. The PPN model is more expressive than SDF, thus it can as well be used effectively to model most streaming applications.

In general Kahn Process Networks (KPNs), of which PPNs represent a special class, are a widely studied distributed model of computation. They are used for describing systems where streams of data are transformed by processes executing in sequence or parallel. Previous research on the use of KPNs in multiprocessor embedded devices has been mainly focusing on the design of frameworks which employ them as a model for application specification [33, 32, 23], and which aim at supporting and optimizing the mapping of KPN processes on the nodes of a reference platform [6, 18]. In [33, 32], different methods and tools are proposed for automatically generating KPN application models from programs written in C/C++. Design space exploration tools and performance analysis are then usually employed for optimizing the mapping of the generated KPN processes on a reference platform. A design phase usually follows in which software synthesis for multi-processor systems [23, 18], or architecture synthesis for FPGA platforms [33] is implemented. A survey of design flows based on the KPN MoC can be found in [17].

The approaches described above, which map applications described as KPNs to customized platforms, have a strong coupling between the application and the platform. Running a different application on the generated platform would not be possible or, even if possible, would give bad performace results. We adopt a different approach where we start by the assumption that we have a platform equipped with (possibly heterogeneous) cores well interconnected with a NoC. We provide a PPN API for this platform that the PPN application processes will comply to. Most importantly, the application code remains the same in all possible mappings of the processes. This is achieved by a proposed intermediate layer, called *middleware*, that includes the mapping related information and implements the PPN communication API.

This approach, where software synthesis relies on the high level APIs provided by the reference platform for facilitating the programming of a multiprocessor system, can be seen elsewhere. The trend from single core design to many core design has forced to consider inter-processor communication issues for passing the data between the cores. One of the emerged message passing communication API is Multicore Association's Communication API (MCAPI) [2] that targets the inter-core communication in a multicore chip. MCAPI is the light-weight (low communication latencies and memory footprint) implementation of message passing interface APIs such as Open MPI [1]. However these MPI standards are not quite fit for the KPN semantics [13] and building the semantics on top of their primitives brings an overhead compared to platforms with dedicated FIFO support.

The communication and synchronization problem when implementing KPNs over multiprocessor platforms without hardware support for FIFO buffers has been considered in [30] and [18]. In [30] the *receiver-initiated* method has been proposed and evaluated for the Cell

BE platform. On the same hardware platform, [18] proposes a different protocol, which makes use of mailboxes and *windowed FIFOs*. The difference with the approach presented in this thesis is that we actually compare a number of approaches to implement the process network semantics, and that we deal with a different kind of platform, with no remote memory access support. Moreover, in both [30] and [18] system adaptivity is not taken into account.

In [31] the problem of implementing the KPN semantics on a NoC is addressed. However, in their approach the NoC topology is customized to the needs of the application at design time and network end-to-end flow control is used to implement the blocking write feature. In this work system adaptivity is considered since the middleware enables run-time management and the platform is generic, i.e. it allows the execution of any application specified as a PPN.

An approach to guarantee blocking write behavior is also used in [4]. That work proposes the use of dedicated operating system communication primitives, which guarantee that the remote FIFO buffer is not full before sending messages through a simple request/acknowledge protocol. The communication approaches described in this thesis assume a more proactive behavior of the consumer processes to guarantee the blocking on write compared to the request/acknowledge protocol. We argue that the presented approach can lead to better performance since it requires less synchronization points.

2.2 Fault-tolerance in NoC-based MPSoCs

As the possibility of experiencing run-time faults becomes increasingly relevant with deep-sub-micron technology nodes, our research activity has been focused on the problem of graceful degradation by dynamic remapping in presence of run-time faults. Within the proposed NoC platform (that will be described in 4, fault-tolerance support has been introduced at both software and hardware levels. The idea is to improve dependability of the system by exploiting the migration method in case of run-time faults in the processing cores. The tasks mapped on faulty cores have to be migrated to fault-free ones at run-time, so that the application can continue its execution without disruption. To this aim, several extensions to the migration mechanism are needed. Firstly, fault detection must be enabled so that the migration can be triggered. Secondly, given that a faulty processor cannot participate in the remapping process, dedicated hardware is needed to ensure the migration functionality to survive in case of malfunctioning. Finally, a remapping decision must be taken in such a way to incur the smallest performance degradation. The details of the proposed solutions are described in Chapter 6.

2.2.1 Open issues and related works

As already said in Section 2.1.1, in [4], a framework that achieves the goals of scalability and system adaptivity is described. Similar to our approach, their platform is programmed using a process network model of computation. However, our approach is fundamentally different because it enables the migration to happen at any time within the main body of the processes. This is a basic requirement in order to allow fault-tolerance, because faults can happen at any time. By contrast, in [4] the process migration is enabled only at fixed points during the execution of processes.

Dynamic task remapping is also performed in [7], [3] by means of a task migration mechanism implemented at user-level or middleware/OS level respectively. Both these approaches require the user to specify checkpoints in the code at which migration can take place. In the presented approach this is not needed because the state that has to be migrated is automatically determined, thanks to the properties of the adopted model of computation (Polyhedral Process Networks [46]).

Task remapping for reliability purposes has been investigated in [24] with the goal of throughput minimization on multi-core embedded systems. The fundamental difference from the presented approach is the use of design-time analysis for all possible scenarios and the storage of all remapping information in the memory. We argue that this technique incurs a large memory requirement to store all fault scenarios.

In [10], a system-level fault-tolerance technique for application mapping, which aims at optimizing the entire system performance and communication energy consumption, is proposed. In particular, the authors address the problem of spare core placement and its impact on system fault-tolerance properties, and propose a run-time fault-aware technique for allocating the application tasks to the available, reachable, and fault-free cores of embedded NoC platforms. In [10], application components running on a faulty core are migrated altogether to available non-employed spare cores, whereas, in our approach, tasks on the faulty core can possibly be remapped to different fault-free cores.

Chapter 3

The MADNESS Project

This chapter features a detailed description of the MADNESS project (Methods for predictAble Design of heterogeneous Embedded Systems with adaptivity and reliability Support), funded by the European Commission. The whole research activity performed over the topics of this thesis, has been driven according to the needs posed by the MADNESS project goals.

The project aims at the definition of innovative system-level design methodologies for embedded MPSoCs, extending the classic concept of design space exploration in multi-application domains to cope with high heterogeneity, technology scaling and system reliability.

The main goal of the project is to provide a framework able to guide designers and researchers to the optimal composition of embedded MPSoC architectures, according to the requirements and the features of a given target application field. The proposed strategies will tackle the new challenges, related to both architecture and design methodologies, arising with the technology scaling, the system reliability and the ever-growing computational needs of modern applications.

The methodologies developed within MADNESS project act at different levels of the design flow, enhancing the state-of-the art with novel features in system-level synthesis, architectural evaluation and prototyping.

3.1 Framework outline

In figure 3.1, a block diagram of the MADNESS system-level design framework is presented. The framework aims at efficiently and effectively performing design space exploration (DSE) to search for the optimal composition of a multimedia NoC-based MPSoC architecture, operating on a library of heterogeneous industrial-strength IP cores and exposing a large number of degrees of freedom.

The MADNESS target platform consists of a library of IP blocks, mentioned in figure 3.1 as *Hardware Library*, which are explored in continuously varying configurations. The project employs a variety of IP building blocks, among which are application-specific instruction-set processors (ASIPs), memories, interconnects, and adapters. This hardware IP library includes industrial-strength blocks from Silicon Hive and Lantiq. Some further blocks, such as a video motion processor, are specifically developed for the MADNESS project. Each ar-

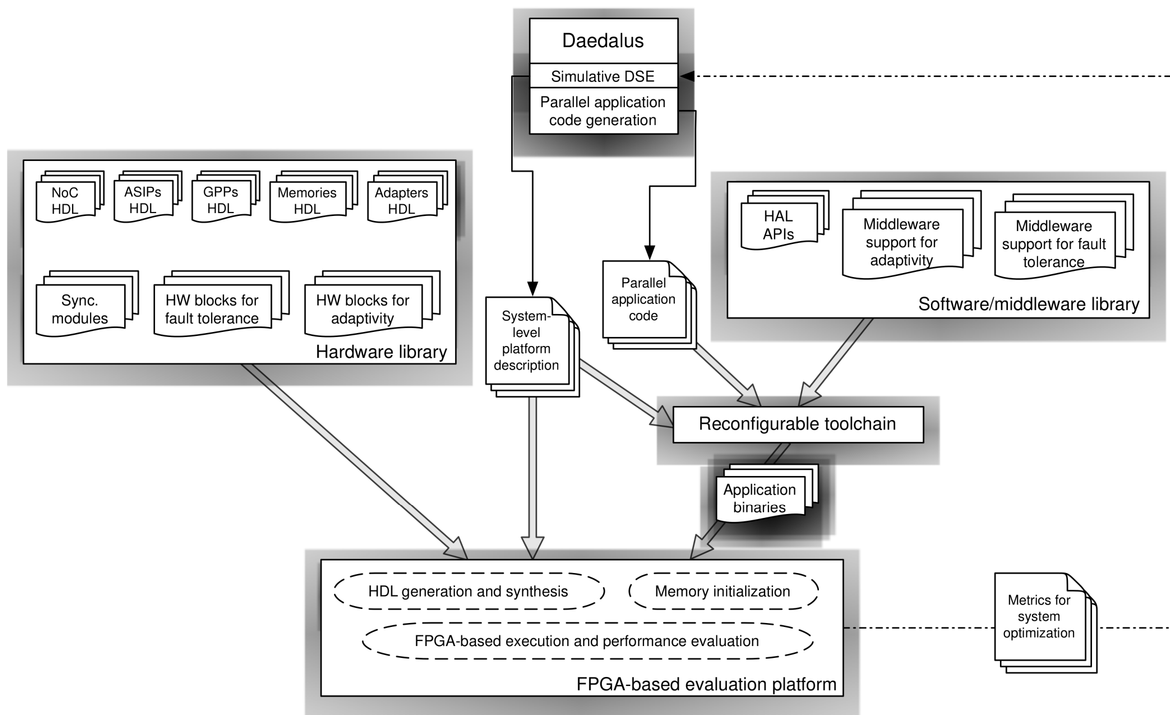

Figure 3.1: The MADNESS system-level design framework for adaptive and fault-tolerant MPSoCs.

chitecture configuration is an abstract collection of target IPs and interconnect structures. The decisions during the optimization process are actually taken by a DSE engine, represented by the box at the top of figure 3.1 and described in detail in section 3.2. The DSE engine iteratively selects one among multiple sets of architecture instances and application mappings, exploiting a specific layer for rapid and accurate architectural evaluation. The DSE engine is based on enhanced versions of several key elements from the Daedalus system-level synthesis design flow [42]. More specifically, it deploys the Sesame simulation environment [37] for simulative DSE and uses the PNgen/ESPAM tools for parallel application code generation [33]. As a consequence, the MADNESS framework, like Daedalus, uses Kahn Process Networks (KPNs) [22] to model parallel multimedia applications. As described later in deeper detail, the Sesame simulation environment has been extended in the scope of the MADNESS project to support DSE for MPSoCs sustaining multi-application dynamic workloads as well as to include with novel techniques for design space pruning including fault-tolerance aspects. As a further point of novelty, the project continuously develops an evaluation layer which integrates a system-level synthesis flow to rapidly evaluate selected design points using an FPGA-based emulation and evaluation platform, described in further detail in Chapter 4. The framework allows to perform system optimization by means of adequately interleaving a high-level simulative design space exploration process with the evaluation of selected design points synthesized on real FPGA-based prototypes. Thus, the DSE engine can access an FPGA-based environment for on-hardware prototyping, when needed during the optimization process, in order to obtain a detailed evaluation of a candidate ar-

chitecture by actually executing the target application on the implemented prototype. In order to improve design predictability, the flow is improved by annotating the emulation results on adequate analytic “technology-aware” models (energy consumption, execution time per frequency, area obstruction). This allows to translate emulation results to a reliable evaluation of a prospective ASIC implementation of the system on a given technology, before actually performing all the back-end fabrication steps. In order to allow the execution of the target application on the FPGA implementation of completely different design points, featuring different kinds of processing elements and interconnects, the framework includes a re-configurable compilation toolchain, depicted in figure 3.1, which is capable of re-targeting itself according to the design point specification. Furthermore, a hardware abstraction layer (HAL) exposes to the programmer a convenient set of APIs that can be used to program the system without referring to specific low-level details of the platform. The compilation toolchain automatically links the appropriate implementation of the API included in the HAL, according to the description of the design point under evaluation. The compilation toolchain and the HAL are described in further detail in section 3.4. Support for system adaptivity and fault-tolerance, and their implementation aspects, which are main topics of this thesis, will be described in Chapters 5 and 6. The implementation of the platform is biased toward low redundancy and power consumption in order to meed the demands of the embedded systems domain. Characteristics of the dynamic behavior and of the resilience to faults can be taken into account by the system-level synthesis during the architectural optimization process, exploiting the mentioned extensions to Sesame.

To allow the mentioned tools and methods to inter-operate without or with minimal manual intervention, the IP-XACT standard [21] was selected for exchanging abstract platform instance descriptions between different tools. However, with regard to this purpose, IP-XACT has shown a number of shortcomings. As a result, several adaptations were made to the IP-XACT standard, allowing us to capture the variability of the target architectures and to capture the power and area consequences of DSE design choices. Finally, in order to actually construct the heterogeneous platform instances needed for FPGA-based evaluation, the MADNESS project has resulted in novel approaches to automatically convert MADNESS IP-XACT descriptions into RTL implementations of multi-ASIP platform instances.

3.2 High-level DSE

The MADNESS framework deploys the Sesame MPSoC simulation framework [37] for simulative DSE. Sesame recognizes separate application and architecture models within a system simulation. An application model, specified as a KPN, describes the functional behavior of a (set of) concurrent application(s). An architecture model defines architecture resources and captures their performance constraints and power consumption characteristics. Subsequently, using a mapping model, an application model is explicitly mapped onto an architecture model (i.e., the mapping specifies which application tasks and communications are performed by which architectural resources in an MPSoC), after which the application and architecture models are co-simulated to study the performance and power consumption consequences of the chosen mapping.

To actually search the design space for optimum design points, Sesame utilizes heuristic search techniques, such as multi-objective Genetic Algorithms (GAs). Such GAs prune the design space by only performing a finite number of design-point evaluations during the

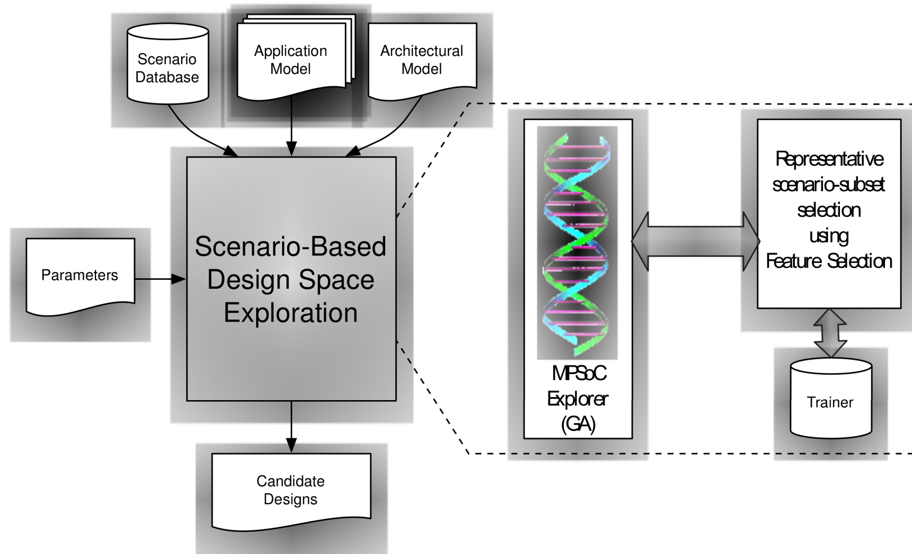

Figure 3.2: Scenario-based DSE.

search, evaluating a population of design points (solutions) over several iterations, called generations. With the help of genetic operators, a GA progresses iteratively towards the best possible solutions.

The Sesame's DSE was focused on the analysis of MPSoC architectures under a single, static application workload. The current trend, however, is that application workloads executing on embedded systems become more and more dynamic. Not only is the behavior of a single application changing over time, but the effect of the interactions between different applications are also hard to predict. This dynamic behavior can be classified and captured using so-called workload scenarios [15]. Workload scenarios make a distinction between two aspects. First, intra-application scenarios describe the dynamic behavior within applications. For example, a QoS mechanism within a decoder application may dynamically lower the bit-rate to save power while still meeting its deadlines. Second, inter-application scenarios describe the interaction between different applications that are concurrently executing on an embedded system and contending for its system resources.

In the context of MADNESS, a novel scenario-based DSE method that allows for capturing the dynamic behavior of multi-application workloads in the process of system-level DSE [43] has been developed. An important problem that needs to be solved by such scenario-based DSE is the rapid evaluation of MPSoC design instances during the search through the MPSoC design space. Because the number of different workload scenarios can be immense, it is infeasible to rapidly evaluate an MPSoC design instance during DSE by exhaustively analyzing (e.g., via simulation) all possible workload scenarios for that particular design point. As a solution, a representative subset of workload scenarios can be used to make the evaluation of MPSoC design instances as fast as possible. The difficulty is that the representative-

ness of a subset of workload scenarios is dependent on the target MPSoC architecture. But since the evaluated MPSoC architectures are not fixed during the process of DSE, we need to simultaneously co-explore the MPSoC design space and the workload scenario space to find representative subsets of workload scenarios for those MPSoC design instances that need to be evaluated. To this end, within MADNESS project a scenario-based DSE method combining a multi-objective GA and a feature selection algorithm have been developed. The GA is used to search the MPSoC design space, while the feature selection algorithm dynamically selects a representative subset of scenarios. This representative subset of scenarios is then used to predict the quality of MPSoC design instances in the GA as accurately as possible.

This scenario-based DSE is depicted in Figure 3.2. As input, the scenario-based DSE uses the application models that need to be mapped onto the MPSoC, an MPSoC platform model, a scenario database in which all possible application scenarios are stored, and search parameters. As output, the DSE produces candidate MPSoC design instances that perform well when considering all the potential situations that can occur in the specified dynamic multi-application workload. The system-level description of the candidate design points is then translated by a utility, and can be elaborated by the FPGA-based prototyping platform, as described in [27].

3.3 FPGA-based evaluation environment

The design flow envisioned in MADNESS raises the need for a fast and accurate evaluation environment. Such a tool has to provide the upper layers of the design flow (i.e., the simulative DSE) with feedback about the performance of any requested candidate architectural configuration. To this aim, within the project, a flexible and fast FPGA-based emulation framework extending the work presented in [28] has been developed. It leverages a library of components, instantiates the desired system configuration, specified through a system-level specification file, and generates the hardware description files for the FPGA synthesis and implementation, automating design steps that are usually very error-prone and effort-hungry. The mentioned feedback, namely consisting of detailed and precise event/cycle-based metrics, is obtained from the execution of the target software application (compiled and linked with the proper toolchains and communication libraries) on the candidate system configuration, implemented on FPGA and adequately instrumented with counters and hardware probes. Moreover, the prototyping environment provides support for *"technology awareness"* within the DSE process, by coupling the use of analytic models and FPGA-based fast emulation. This allows to obtain early power and execution time figures related to a prospective ASIC implementation, without the need to perform long post-synthesis software simulations. The FPGA emulation results are back-annotated using analytic models for the estimation of the physical figures of interest. Timing results (cycle counts) are evaluated according to the modeled target ASIC operating frequencies and the evaluated switching activity is translated into detailed power numbers. Thus, the assumptions made at the system-level design phase can be verified before the actual back-end implementation of the system, increasing the overall convergence of the design flow. The models included in the evaluation platform are built by interpolation of layout-level experimental results obtained after the ASIC implementation of the reference library IPs, along the lines already defined for NoC building blocks in [26]. In the latter, the accuracy of the models is assessed to be higher than 90% when complete topologies are considered, with respect to post-layout analysis of real

ASIC implementations. A detailed description of the evaluation platform will be provided in Chapter 4.

3.4 Compilation toolchain and hardware abstraction layer

This section describes the interaction between the DSE, the compilation toolchain and the evaluation platform. Further, a brief overview of the extensions to the available compilers is given. Finally the key features of the hardware abstraction layer and their integration in the compilation toolchain are described.

3.4.1 Compilation toolchain

The DSE tool passes the sources for each process and additional information like mapping and system description to the compilation toolchain (see Figure 3.1). The compilation toolchain takes care of the correct mapping between processes and their corresponding compilers for each processor in an automatic way. In order to determine the right compiler, the user needs to pass an environment description to the compilation toolchain. This description defines which compilers are available and which options to use for each processor in the system.

To meet the requirements of the framework, a hardware abstraction layer (HAL) is integrated in the compilation toolchain in a retargetable manner.

3.4.2 Hardware abstraction layer

The MADNESS framework aims at generating an optimal MPSoC for given hardware components and for a given application. Therefore, the application developer does not know the platform during the development process. Special processor-dependent instructions cannot be exploited during system generation. Even the realization of simple low-level functions like communication or synchronization without knowledge of the underlying processors and hardware components is impossible. For this reason, the MADNESS framework provides a Hardware Abstraction Layer (HAL) which enables the developer to create portable and processor-dependent optimized applications. The developed HAL consists of two parts, a fixed part which covers the standard abilities of processors and a generic processor-dependent part.

Standard abilities

Standard abilities include memory access, communication or synchronization mechanisms. One assumes that present and future processors support these mechanisms. As a consequence MADNESS defines a fixed set of standard functions which have to be implemented for each available processor used in the proposed framework.

Special abilities

The actual advantage of heterogeneous multi-processor-systems is the composition of different specialized processors. To achieve the best performance one has to take into account

that the HAL has also to cover the processor-dependent features. These features are called special ability functions. This part of the HAL is generic and extensible by the user without modifications of the used compilers. For each specialized implementation, a standard ANSI-C implementation must be provided. Thus, the availability of a semantically equivalent application, if the corresponding processor feature is not available, is ensured.

3.5 Support for Adaptivity

The term system adaptivity refers to the ability of a system to dynamically assign tasks of the application(s) running on it to the resources available over time. This is an emerging topic in MPSoC design due to recent evolutions in embedded systems [34]. The KPN model of computation (MoC), adopted in the MADNESS framework to model multimedia applications for mapping them onto the MPSoC, presents remarkably simple operational semantics and distributed control, which allow for a natural realization of system adaptivity mechanisms.

From an architectural point of view, the framework is focused on tile-based NoC [12] systems. Among other advantages, this choice is driven by the goal of system adaptivity. NoC-based interconnects' flexibility allows to overcome the drawbacks exposed by point-to-point connections classically used in multimedia. Point-to-point connections are typically more efficient in terms of communication latency, but they are intrinsically less efficient in supporting communication patterns varying at runtime, unless full connectivity among all the processors in the system is provided at design time, at the price of making wiring and buffering of the whole communication structure rather complex. Moreover, NoC communication infrastructures are physically and functionally more scalable than bus-based shared memory systems [5].

The starting assumption is that the target platform is equipped with a heterogeneous set of cores interconnected with a NoC. The application code must remain the same in every possible mapping of the tasks to allow for system adaptivity. This fact implies that the used communication primitives must be neither platform dependent nor mapping dependent. An intermediate layer, or *middleware*, has been implemented to refine such communication primitives, including mapping related information, and to respect the KPN semantics on the NoC-based MPSoC platform.

The middleware layer and the proposed adaptivity strategies will be thoroughly discussed in Chapter 5.

3.6 Support for Fault-tolerance

Applications of embedded systems increasingly require high availability of the systems themselves, possibly accepting a measure of graceful degradation. Moreover, increasing complexity of the systems is reaching such levels that the probability that some manufacturing defect will escape end-of-production testing or that faults will become evident during normal operation has to be taken into account. Standard approaches based on massive redundancy are not directly applicable to embedded platforms, constrained by the need for solutions having low cost and low power consumption. New approaches are therefore needed.

The MADNESS project focuses on the development of fault tolerant solutions which are not dependent on a technology-related low-level fault model, but rather on technology-

abstracting functional-level error models. This approach allows the development of a functionally identical system for two different implementation technologies - FPGA and ASIC - such that the system's evaluation on one technology can be immediately adoptable and credible for the other technology.

The fault tolerant approaches focus on the detection of run-time faults and on the use of reconfiguration strategies implemented at different levels. In the MADNESS framework, three main types of components are taken into account, i.e., *processing cores*, *storage elements*, and the *network-on-chip* (NoC). While for storage components standard fault tolerant strategies based on error detecting and correcting codes are adopted, for the NoC and the processing elements ad-hoc strategies for fault detection and reconfiguration have been developed, and will be presented in Chapter 6.

Chapter 4

The MADNESS evaluation platform

4.1 Platform overview

As described in Chapter 3, within the MADNESS project, an integrated framework for the application-driven design of MPSoCs was studied and implemented, aimed at supporting the designer during such a complex process. Figure 4.1 gives a block decomposition the MADNESS tool support for prototyping activity, which is based on the SHMPI system-level FPGA-based prototyping environment, presented in [39]. A system-level platform description is input to the framework. The description includes processing, interconnection and memory modules instantiation and configuration, and address space partitioning.

A high-level topology compiler is in charge of parsing the topology description file and generating the RTL files that describe the hardware top view of the complete platform. This stage is also intended to generate the hw/sw platform description files to be passed to the Xilinx platform instantiation toolset. This phase of the flow will be specifically addressed and further explained in Section 4.4. The composition and configuration of the selected platform builds upon a repository of soft IP cores. The content of this library will be described in detail in Section 4.3. The use of these repositories does not prevent the inclusion of further modules into the system, since custom cores can always be added as RTL or pre-synthesized netlists with little effort. This is a crucial feature of the RTL libraries, since extensibility is key to the applicability range of the framework. That is also the reason why all the modules included in the library are fully compliant to a common interfacing standard, which is a subset of the well-known OCP-IP communication standard [35], as we will discuss in the following sections.

Regarding the software part of the system, the Xilinx development environment includes the standard compilation and debugging tool-chains for the soft processors that can be instantiated, while the drivers of the peripherals are automatically generated already at the platform compilation stage, according to the parameters provided by the user. The RTL files of the components can potentially be passed, with minor modifications, to an ASIC synthesis flow, in order to obtain power and area figures of the designed platform and to refine the area and power models of the building blocks instantiated within the system. So far, this capability is provided only for the interconnection modules. The framework operating flow proceeds with the FPGA synthesis and implementation through the adoption of the Xilinx proprietary tools (within the Xilinx ISE©environment). The execution of the targeted appli-

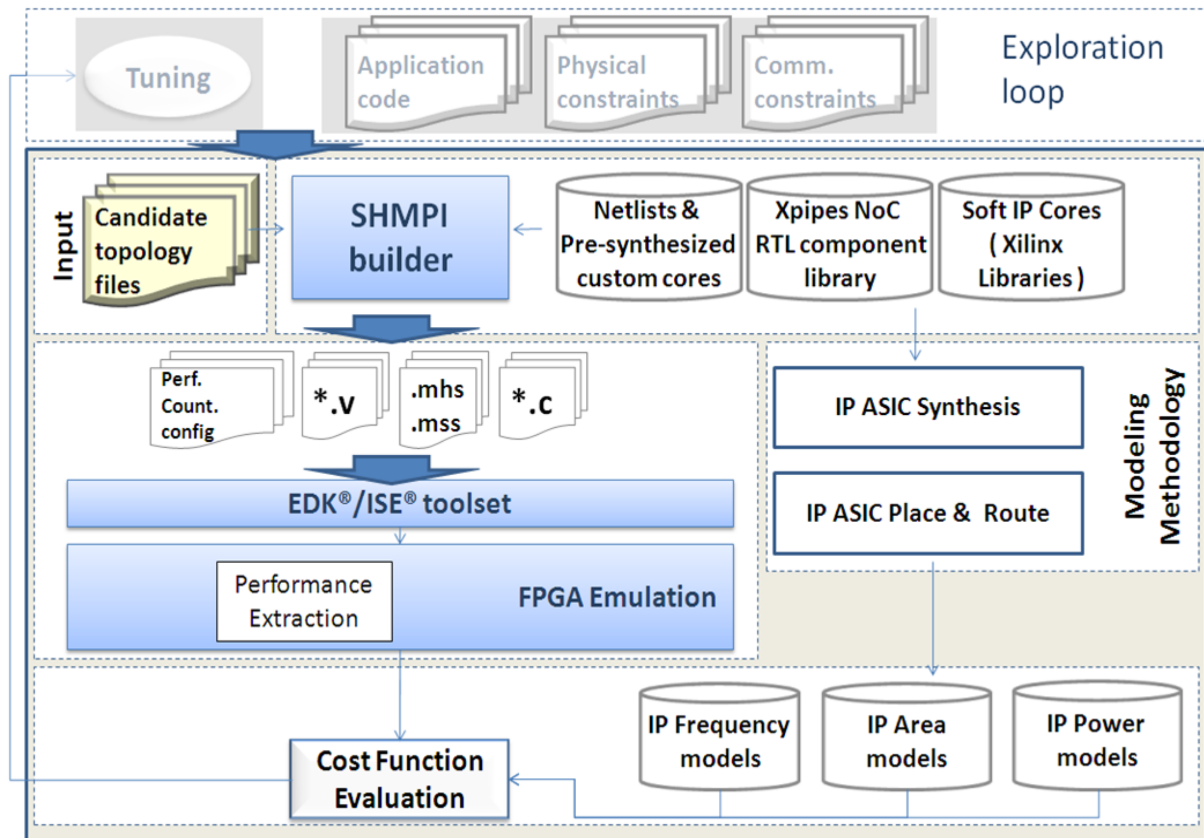

Figure 4.1: Overview of the evaluation platform

cation on the configured FPGA can be easily profiled with deep accuracy. Moreover, the emulation of the complete platform enables the rapid collection of cycle-accurate information on the switching activity, that can be used, in cooperation with the available power models, to obtain detailed figures related to a prospective ASIC implementation of the system.

4.2 System-level platform description input file

The designer is able to input a platform description by passing as input to the framework a text file that describes the main system-level building blocks of the design under emulation. The granularity of the input file is at the single processing, interconnection, I/O and memorization module level. Regarding the interconnection subsystem architecture, the designer can describe, in case a source routing NoC-based is selected (which is the case of the ×pipes interconnection library), the number of switches, the n-arity of each switch, the number of buffering input/output stages, the routing tables and normally has to tag the different links connecting the switches. Regarding the memorization layer, the type of memory has to be selected. Currently the system is able to instantiate on-FPGA Xilinx proprietary hard BRAM modules, configuring them to emulate single- and double-port memory cores. The address space and the related processor into which the memory module is connected have to be specified. If the memory is shared among the different processing element, the module has to be declared to be such. Regarding the different I/O and synchronization controller and modules, the address spaces have to be declared as well.

The following code contains snippets taken from an actual input system description file:

```
// -----------------------------
// define the topology here
// name, mesh/torus specifier (mesh/torus/other)
// -----------------------------
topology(topology_2x2, other);

// -----------------------------
// define the cores here
// core name and number, switch number, NI clock divider, NI buffers,
// initiator/target type, type of core (if a specific one is requested),
// memory mapping (only if target), fixed specifier
// (only if target and of shared type)
// -----------------------------
core(core_0, switch_0, 1, 6, userdefined, initiator);
core(core_1, switch_1, 1, 6, userdefined, initiator);
core(core_2, switch_2, 1, 6, userdefined, initiator);
core(core_3, switch_3, 1, 6, userdefined, initiator);

core(pm_4,   switch_0, 1, 6, double, target:0x10,high:0x1000ffff);
core(pm_5,   switch_1, 1, 6, double, target:0x12,high:0x1200ffff);
core(pm_6,   switch_2, 1, 6, ocpmemory, target:0x14,high:0x1400ffff);
core(pm_7,   switch_3, 1, 6, ocpmemory, target:0x16,high:0x1600ffff);

core(ts_8,   switch_3, 1, 6, Testandset,target:0xff,high:0xffffffff);
core(ul_9,   switch_0, 1, 6, Uartlite,target:0x46,high:0x4600ffff);
core(shm_10, switch_1, 1, 6, shared, target:0x06, high:0x0600ffff);

// -----------------------------
// define the switches here
// switch number, switch inputs, switch outputs, number of buffers,
// core ID to which the switch performance counter is attached,
// port ID to which the switch performance counter is attached.
// -----------------------------
switch(switch_0, 5, 5, 6, 0, 0);
switch(switch_1, 5, 5, 6, 1, 0);
switch(switch_2, 5, 5, 6, 2, 0);
switch(switch_3, 5, 5, 6, 3, 0);

// -----------------------------
// define the links here
// link number, source, destination
// -----------------------------

link(link0,  switch_0,  switch_1);
link(link1,  switch_1,  switch_0);
```

```
.
.
.

link(link6,  switch_1,  switch_3);
link(link7,  switch_3,  switch_1);

// ----------------------------
// define the routes here
// source core, destination core, the order in which switches need to be
// traversed from the source core to the destination core
// ----------------------------
route(core_0, pm_4,   switches:0);
route(core_0, ts_8,  switches:0,1,3);
route(core_0, ul_9,  switches:0);
route(core_0, shm_10, switches:0,1);

.
.
.

route(core_3, pm_7,   switches:3);
route(core_3, ts_8,  switches:3);
route(core_3, ul_9,  switches:3,2,0);
route(core_3, shm_10, switches:3,1);

route(pm_4,  core_0, switches:0);
route(pm_4,  pm_5, switches:0,1);
route(pm_4,  pm_6, switches:0,2);
route(pm_4,  pm_7, switches:0,1,3);

.
.
.

route(pm_7,  core_3, switches:3);
route(pm_7,  pm_4, switches:3,2,0);
route(pm_7,  pm_5, switches:3,1);
route(pm_7,  pm_6, switches:3,2);

.
.
.

route(ts_8,  core_0, switches:3,2,0);
route(ts_8,  core_1, switches:3,1);
```

```
route(ts_8,  core_2, switches:3,2);
route(ts_8,  core_3, switches:3);
```

The code snippet contains the description of a sample topology named `topology_2x2`, which contains 4 processing elements, 2 double-port local memories, 2 single-port local memories, a test&set synchronization module, an UART controller for interfacing with a serial port and a shared memory, all declared with the same `core()` primitive and different parameters. The parameters allow for specifying different clock domains (unused in this case), the number of buffering stages in the network interface (6 in this case), the core identifier and the basic memory mapping of each device. Local memories with multiple ports have different mappings for each port with respect to the local processor and the other processors, if direct messaging is enabled.

The NoC switches are defined using the `switch()` primitive, whose parameters allow the designer to specify a name for the switch, the number of inputs and outputs of that switch, which define the arity of the switch itself, the number of buffering stages (output buffering is used in this interconnection library module) and two other IDs. These two other IDs respectively identify the core and the port which the switch performance counters are actually attached to. The actual performance counter logic will be described in detail in Section 4.5. The `link()` primitive enable link identification. Its parameters simply point to the interconnected switch modules. Links have to be declared as if they were half-duplex, meaning that between two switches connected through a full-duplex link there must be two separate links. In addition to that, if source routing is used, the `route()` primitive enables specifying the routing path between every core pair inside the network, and its parameters specify the list of switch traversed by the packets from source to destination. Direct communication between two computing cores is not allowed. If direct messaging is intended to happen at higher level, the underlying mechanism implies a communication from the source processing element to the remote memory that is attached to the destination processing element.

4.3 The soft IP cores RTL repository

This section will describe the components currently available in the soft IP cores repository. All the modules included in the libraries are highly parametric. This is a key feature with respect to the prospective adoption of the framework for design exploration purposes. Different computing and memorization elements have been added to the libraries and they will not be described within this thesis. The building blocks have been made compliant, where not already done, to a subset of the well known OCP open communication standard [35]. We will first provide a brief introduction to the main features of the standard.

4.3.1 Elements of the Open Core communication Protocol (OCP)

The Open Core Protocol (OCP) defines a bus-independent protocol between IP cores that reduces design time, design risk, and manufacturing costs for SOC designs. An IP core can be a simple peripheral core, a high-performance microprocessor, or an on-chip communication subsystem such as a wrapped on-chip bus. The OCP defines a single-clock synchronous interface point-to-point interface between two communicating entities such as IP cores and

bus interface modules (bus wrappers). One entity acts as the master of the OCP instance, and the other as the slave. Only the master can present commands and is the controlling entity. The slave responds to commands presented to it, either by accepting data from the master, or presenting data to the master. For two entities to communicate in a peer-to-peer fashion, there need to be two instances of the OCP connecting them - one where the first entity is a master, and one where the first entity is a slave. The characteristics of the IP core determine whether the core needs master, slave, or both sides of the OCP.

A transfer across this protocol occurs as follows. A system initiator (as the OCP master) presents command, control, and possibly data to its connected slave (a wrapper interface module that abstracts the underlying hardware details). The interface module plays the request across the on-chip physical interconnection system, which can be a bus, a crossbar, a NoC or whatever. The OCP does not specify the underlying hardware functionality. Instead, the interface designer converts the OCP request into a transfer for the underlying hardware. The receiving wrapper interface module (as the OCP master) converts the underlying hardware transfer operation into a legal OCP command. The system target (OCP slave) receives the command and takes the requested action.

The OCP standard defines a huge number of signals, which are grouped in different categories, according to the functionality that the different interconnected modules provide. We chose to implement only compliance with a small subset of the OCP signals, whose list is reported in Table 4.1 together with a brief signal functionality description.

Name	Width	Driver	Function
Clk	1	varies	OCP clock
MAddr	configurable	master	Transfer address
MCmd	3	master	Transfer command
MData	configurable	master	Write data
MDataValid	1	master	Write data valid
MRespAccept	1	master	Master accepts response
MByteEnable	4	master	One-hot byte enable
SCmdAccept	1	slave	Slave accepts transfer
SData	configurable	slave	Read data
SDataAccept	1	slave	Slave accepts write data
SResp	2	slave	Transfer response

Table 4.1: Implemented OCP signals

Where needed, OCP-compliant wrappers modules have been developed to adapt the original module interface to the one specified by the protocol.

4.3.2 Processing elements, memories and I/O

The computing element library has been mainly focused around the FPGA-oriented Xilinx proprietary soft cores, that is to say the Microblaze©[47] and PowerPC©[48] cores. Within MADNESS project, a set of other processing elements have been added to the repository, namely a set of multi-media oriented Application Specific Integrated Processors (ASIP) provided by Intel Corporation, and a processor specifically designed for the packet processing domain provided by Lantiq Deutschland GmbH.

As for what regards the memorization and I/O elements, standard Xilinx soft cores are currently being used. The memory modules can be automatically configured as single- or double-port BRAM-based modules. The separation between such cases is operated according to the system-level description provided by the designer. In case direct processor-to-processor message-passing is enabled, double-port memories will be selected, together with the necessary logic to handle DMAs at the network interface level.

Regarding the I/O, standard controller modules are available, such as UART, Ethernet and DVI/TFT controllers.

4.3.3 Interconnection elements

The ×pipes NoC component library ([11]) is a highly flexible library of component blocks that has been chosen as baseline reference for the development activity. The library is suitable for the creation of arbitrary topologies, thanks to the capability of its modules of being almost completely configured at design time. ×pipes, natively, includes three main components: switches, network interfaces (NIs) and links. Figure 4.2 plots the basic architecture of the ×pipes switch. It is a very simple switch configuration, where output buffer is employed through multi-stage variable-latency FIFOs. A round-robin priority arbiter with selectable inputs is employed to allocate the all-to-all crossbar output ports. The minimum traversal latency per switch is 2 clock ticks per flit.

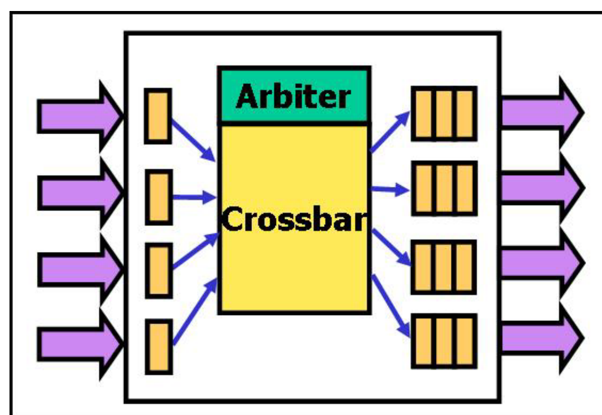

Figure 4.2: ×pipes switch architecture

The backbone of the NoC is composed of switches, whose main function is to route packets from sources to destinations. Arbitrary switch connectivity is possible, allowing for implementation of any topology. Switches provide buffering resources to lower congestion and improve performance. In ×pipes, both output and input buffering can be chosen, i.e. FIFOs may be present at each input and output port. Switches also handle flow control issues, and resolve conflicts among packets when they overlap in requesting access to the same physical links. A NI is needed to connect each IP core to the NoC. NIs convert transaction requests/responses into packets and vice versa. Packets are then split into a sequence of flits before transmission, to decrease the physical wire parallelism requirements. In ×pipes, two separate NIs are defined, an initiator and a target one, respectively associated to OCP system masters and OCP system slaves. A master/slave device will require an NI of each type to be attached to it. The interface among IP cores and NIs is point-to-point as defined by the OCP

subset described in Table 4.1, guaranteeing maximum reusability and compliance with the interface standards.

NI Look-Up Tables (LUTs) are used to specify the path that packets will follow in the network to reach their destination (source routing). Two different clock signals can potentially be attached to the NIs: one to drive the NI front-end (OCP interface), the other to drive the NI back-end (×pipes interface). The ×pipes clock frequency must be an integer multiple of the OCP one. This arrangement allows the NoC to run at a fast clock even though some or all of the attached IP cores are slower, which is crucial to keep transaction latency low. Since each IP core can run at a different frequency of the ×pipes frequency, mixed-clock platforms are possible. Inter-block links are a critical component of NoCs, given the technology trends for global wires. The problem of signal propagation delay is, or will soon become, critical. For this reason, ×pipes supports link pipelining, i.e. the interleaving of logical buffers along links. Proper flow control protocols are implemented in link transmitters and receivers (NIs and switches) to make the link latency transparent to the surrounding logic. Therefore, the overall platform can run at a fast clock frequency, without the longest wires being a global speed limiter. Only the links which are too long for single-cycle propagation will need to pay a repeater latency penalty.

Within the development of the evaluation framework, the original ×pipes library has been extended explicitly for adaptation and integration in MADNESS project, and to provide advanced communication capabilities required for fast prototyping. Here follows a list of the main features that have been added to the library:

- Capability of initializing and handling DMAs (meaning direct memory to memory transfers). The need for this feature has appeared in order to support, at low level, all those models of computations that rely on direct processor-to-processor message passing. In order to implement this added capability, additional logic has been inserted in the processor and memory network interfaces. The way this logic works is basically that the sending processor programs, through memory-mapped registers, a DMA transfer from its memory to a destination memory, by specifying the destination network address and the burst length. The transaction is then translated into an OCP burst transfer, that takes place from the source memory directly to the destination one. Upon receive, the destination network interface is able to store the incoming data on a temporary memory buffer or, if the receiving processor has already reached the receiving primitive call within the application, directly into the destination memory area. Further detail on the software implementation of the message-passing strategy will be provided in Section 4.3.7.

- Insertion of performance counters inside NoC modules has been enabled through addition of dedicated hardware monitors directly attached to the output buffers of the switch. The value of the counters are then written into dedicated memory-mapped registers through which they are accessible to the processing element.

All the mentioned additional features required some modifications of the processor-to-NI interface circuitry. Several dedicated adapters have been developed in this aim, allowing at the same time the seamless integration of the ×pipes library (natively compliant with OCP) with the rest of the environment. Some address decoding logic has been added inside the core in order to detect those load/store operations that are not intended to generate

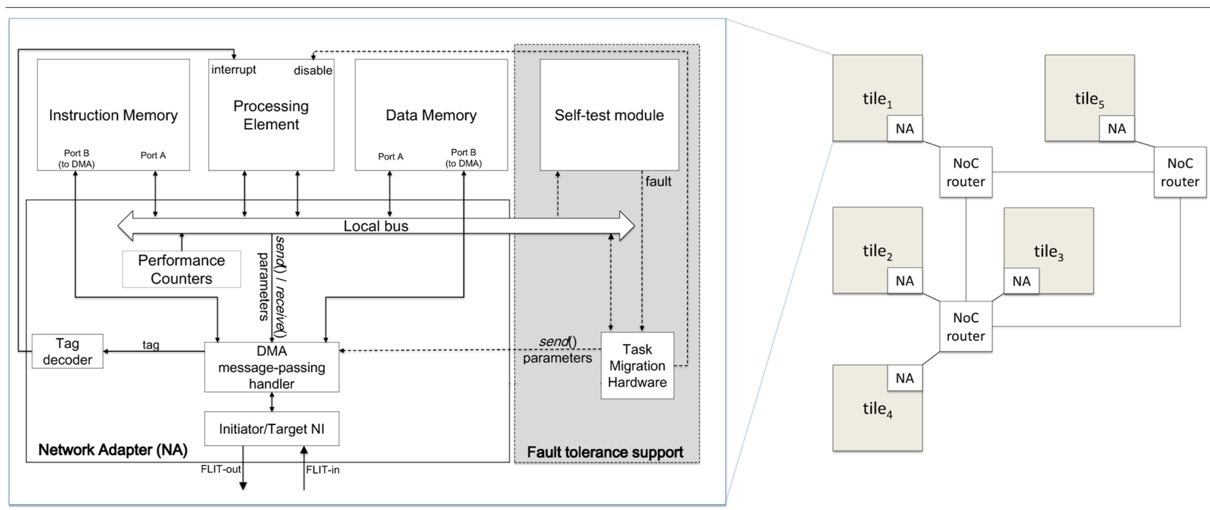

Figure 4.3: A general overview of an example template instance

traffic over the network, such as accesses to memory mapped registers or to performance counters.

4.3.4 Message Passing programming model

Reference primitives implementing message-passing communication are built, according to the general definition of such model, upon two base functions: *send()* and *receive()*. These two primitives are implemented in C, and interact with the hardware structures described in Section 4.3.5. According to the usual message-passing signatures, to send a message with a *send()*, the programmer has to specify the address (*SendAddress* hereafter) inside the private memory that contains the information to be sent (message data), a tag assigned to the message (*SendTag*), the size of the transfer (*SendDim*), and the ID of the destination processor (or process, in case of multi-context execution in the processing elements - *SendID*). The *receive()* parameters are the tag of the expected message (*ReceiveTag*), the sender ID (*ReceiveID*) and the address where the received message data has to be stored (*ReceiveAddress*). Two implementations of the *receive()* are provided, with blocking and non-blocking behaviour.

4.3.5 Message Passing hardware support

The *Network Adapter* architecture is depicted in Figure 4.3 (left side). Both the instruction and data private memories of the processor have two access ports, in order to allow the processor to keep on accessing code and data from one instruction and one data port, while, at the same time, the other ports can be used to directly load/store data from/to the memory in case of message send/receive. In this way, communication and computation can overlap, potentially leading to a significant speed-up. The *NA* integrates a local bus, that, according to the address requested by the processor interface, enables access to:

- the private memory,
- a module called *DMA message-passing handler* (MPH),

- a set of performance counters to obtain statistics about the application execution

In the figure, the gray part represents the additional circuitry supporting fault-tolerance, that will be described in Chapter 6. The MPH embeds a set of memory-mapped registers that are programmed by the processor, to control send and receive operations, setting the previously described parameters.

It also includes an address generator in charge of generating the addresses when the private memories must be accessed from the port reserved for message passing.

When the processor wants to call a *send()*, the microcode that implements the primitive stores the required values into the send-related memory-mapped registers. As soon as the registers are programmed, the *address generator* starts to load *SendDim* words from the memory, starting from address *SendAddr*, and propagates them to the NI. The destination address requested for the network transaction is obtained by the *address generator* according to the content of *SendID*, translating the destination process ID into the network address of the destination processor private memory.

At the other end of the communication, the processor needs to execute a *receive()* to complete the transaction. It may happen that the *receive()* has not been called at the moment the packets composing the message actually arrive to the destination network node. In this case the message data is stored in the memory, inside a (configurable) memory buffer reserved for such a purpose. The identification fields related to the incoming message (sender, tag, buffer address) are stored inside an event file, in order to enable the *receive()* primitive to retrieve the message from the memory when it will eventually be executed. The *receive()* microcode, as a first step, stores the parameters inside three memory-mapped registers. Once such registers are programmed, the processor must keep accessing the DMA, scanning the event file locations, to check if the message under reception is already inside the buffer. In the case of a match, the processor copies the message data from the buffer to the *ReceiveAddress*. If the message is not found in the event file, the processor keeps polling the DMA handler, where a dedicated circuitry is in charge of comparing the incoming messages with the contents of the three registers. In case of matching, the message data is stored in memory, directly at the location identified by *ReceiveAddress*. In order to allow partial buffer de-fragmentation, the buffer is treated as a list.

4.3.6 Interrupt generation support

A tag decoder has been instantiated inside the Network Adapter. It is in charge of detecting a set of pre-determined tag configurations, that are reserved for the purpose of remote interrupt generation. In case of matching, the tag decoder triggers an interrupt signal that is connected to the processor interrupt controller. This feature can be used to allow a processor in the system to generate an asynchronous event on another processor, such as the initiation of the migration process.

4.3.7 Software libraries

As part of the library-based approach, we developed several software routines that constitute the framework Hardware Abstraction Layer (HAL). These routines are also required to provide to the application/firmware level the necessary APIs to implement the shared-memory

model of parallel computation, and can be included through standard header files. Here follows a list of the main software functions, along with their functionality:

- *Shared memory lock/unlock primitives* - these functions will be described in Section 4.3.8. They provide lock/unlock primitives for shared-memory multi-core systems and rely on the hardware *Test&Set* synchronization module.

- *Thread spawn and wait primitives* - these functions emulate the creation of threads of execution on remote processors. The thread creation is emulated through pointer passing. Basically, the spawning thread calls a `create()` function, that sets a pointer in a shared memory location and wakes up a remote processor that was polling on that address. This mechanism, however, has the strong implication that all the processors load the same instruction and initialized data regions at startup, in a SPMD fashion. In such a way that thread instructions are already located in each processor local instruction memory. Otherwise, remote thread creation would have implied a thread load from the shared memory or, more likely, a thread load from the off-chip memory. The spawned thread, as already hinted, runs on a processor that was stuck waiting for the function pointer on that specific location, by calling a `wait_task()` function. Upon exit, the `wait_task()` function returns the pointer to the task to be executed.

- *Shared memory barrier synchronization primitives* - these functions implement the barrier synchronization for shared memory systems. They rely on the specific `bar_type` data type, whose struct follows:

  ```
  typedef struct bar_type_ {
  volatile int num_p;
  volatile int counter;
  volatile int lock_b;
  volatile int flag;
  } bar_type ;
  ```

  The barrier is implemented by atomically (locking the `lock_b` variable) incrementing a shared counter (`counter`) and by busy waiting until it reaches the predefined number of accesses (`num_p`), meaning that all the desired threads have entered the barrier. The barrier has to be first initialized by calling the `barinit()` function, that specifies the locking address and the number of expected threads.

- *printing functions* - these functions have been designed for debug purposes and print characters and numbers through the UART controller to the serial I/O port. Atomic use of the controller is guaranteed both at the single character level and at entire string level. Their names are `shmpi_print()` and `shmpi_putnum()`.

- *functions for accessing the performance counters* - these functions access the memory-mapped performance counters and print them on the UART controller for debugging and elaboration purposes. There are separate functions for accessing the performance counters related to processing/memorization elements, called `print_core_pc()` and for the switching elements, called `print_switch_pc()`. More details on the performance counters will be provided in Section 4.5.

4.3.8 Shared memory support: synchronization modules

Every application written to exploit thread-level parallelism on top of a multi-core platform, not only in the embedded field, requires synchronization among the different running threads. In case the memory organization includes some kind of shared memory layer, this can be implemented through well-established lock-unlock mechanisms. These mechanisms can be implemented through full software solutions that rely on specific atomic instructions and ISAs, or through dedicated hardware modules that handle the atomicity control. In order to implement atomic memory access to specific locations, since not all the processors included in the processing element library supported native LL/SC instructions within their ISAs, we decided to develop hardware semaphore modules for locking/unlocking specific memory locations. These modules implement in hardware the atomic *Test&Set* mechanism.

The basic idea of such mechanism is, before actually accessing the shared memory location, to acquire a hardware lock for that location. The hardware lock is asked for acquisition via a simple load to a specific memory address (the address of the semaphores bank). Upon request, the bank of semaphores has specific logic to check if a request for that location has already arrived, and if not, replies with an ACK (encoded in the data field) and atomically locks that location. If that location is already locked, on the opposite, the reply data will contain a particular encoding for a NACK. The hardware implementation for such a mechanism requires a bank of registers (the lock registers) to store the requested location addresses and the semaphore bits (one for each location), plus all the logic to compare in a combinatorial fashion the desired address with all the locked addresses. The hardware *Test&Set* has been implemented with a standard OCP interface and can be added as a normal I/O module, by specifying its memory-mapping.

On the software side, two low-level functions have been implemented to lock and unlock a specific location. The functions have been implemented as assembly microcode to optimize the execution time. Their code is structured as follows, considering a network address for the *Test&Set* modules of $0xFF$ in the address MSB:

```
void lock(int * lock_index){

__asm__ (
"addi r1, r1, -12 \n\t "
"  sw r10, r0, r1  \n\t "
"  swi r9, r1, 4  \n\t "
"  swi r11, r1, 8  \n\t "
" or r0, r0, r0  \n\t "
" ori r9, %0, 0xff000000  \n\t "
" LOCK: \n\t "
" lw r10, r0, r9  \n\t "
" or r0, r0, r0  \n\t "
" or r0, r0, r0  \n\t "
" bnei r10,LOCK  \n\t "
" or r0, r0, r0   \n\t "
" or r0, r0, r0   \n\t "
      "  lw r10, r0, r1  \n\t "
      "  lwi r9, r1, 4  \n\t "
```

```
      "  lwi r11, r1, 8  \n\t "
      "  addi r1, r1, 12  \n\t "
:
:"r" (lock_index) //input
);
}

void unlock(int * lock_index){

__asm__(" UNLOCK:      "
" addi r1, r1, -8 \n\t "
"  sw r10, r0, r1  \n\t "
"  swi r9, r1, 4  \n\t "
" or r0, r0, r0  \n\t "
" ori r9, %0, 0xff000000  \n\t "
" sw r0, r0, r9  \n\t "
" or r0, r0, r0  \n\t "
" or r0, r0, r0  \n\t "
"  lw r10, r0, r1  \n\t "
      "  lwi r9, r1, 4  \n\t "
      "  addi r1, r1, 8  \n\t "
:
: "r" (lock_index) //input
);
}
```

4.4 The SHMPI platform builder

As illustrated in Figure 4.1, the platform instantiation stage is handled by the SHMPI topology compiler, a tool that automatically creates the hardware/software platform description files basing on the specification input file provided by the user, instantiating the desired set of building modules (cores, interconnection building blocks, memories) from the library of configurable soft-cores. The SHMPI topology compiler extends to the composition of the entire multi-core hw/sw platform and to the integration with the Xilinx development tools the ×pipes compiler, a tool developed for the automatic instantiation of application-specific interconnection networks [11]. In further detail, the SHMPI topology compiler able to construct the desired platform, instantiating and interconnecting, through a customized ×pipes NoC layer, an arbitrary number of processors, memories (private or shared), memory controllers, I/O peripherals, buses, bridges, dedicated point-to-point communication channels, etc.

The topology compiler automatically generates the hardware/software description files that are necessary for the FPGA implementation of the whole hw/sw platform. Since the Xilinx proprietary tools are used to handle the FPGA synthesis flow, the generated files must respect the specific syntax in order to be correctly processed.

From an implementation viewpoint, the platform builder has been developed as a stand-

alone sequential C/C++ code, which runs through different consecutive phases, as described in the following list:

- a parsing phase, which scans the entire system-level description input file to build a set of data structures that store the number and identifiers of cores, memories, interconnection elements, links and routing tables.
- a switch configuration phase, which calls a different program to configure the switch according to the designer description and eventually generate their RTL description. This phase also generates the routing tables initialization phase in order to correctly implement source routing within the network.
- a *top module* generation phase, which builds the RTL description of the top level Verilog platform description file. This file contains the highest view of all the modules included in the system to be synthesized for FPGA.
- a phase that generates all the files necessary to the Xilinx proprietary FPGA synthesis and implementation flow.
- a simulation script generation phase. This phase generates a ModelSim script file (*.do*) and the related testbench file (*.v*) in case software waveform simulation of the platform needs to be performed for debugging purposes.
- a phase to generate the memory initialization files (*.bmm*), that direct the Xilinx toolchain in correctly mapping the application binaries into the different BRAM modules instantiated in the platform.

Upon successful execution of the SHMPI builder, all the directories and files are in place for continuing the FPGA implementation flow down to device configuration. These phases rely on Xilinx proprietary toolchain, therefore require its availability for the designer. If no manual tuning is needed at this stage, the toolchain can be traversed with a single script which is available for the designer.

4.5 Performance extraction

The extraction of the performance metrics is handled through the insertion of a dedicated set of event-counters, directly connected to the monitored logic. The insertion of this measurement subsystem does not overload the whole emulation in terms of occupied hardware resources within the FPGA fabric, since the event-counters involve very scarce logic utilization. Three types of performance counters are allowed, according to the architectural element they are connected to. It is possible to insert monitors at the processing core interface, at the switch output channel interface and at the memory port interface. The specification of which events are intended to be monitored can be easily included in the topology file that is passed as input to the whole framework. The MADNESS topology compiler is able to handle the insertion of the necessary hardware modules and the automatic binding of the event-counters to the proper signal.

The basic usage of the event-counters does not imply the instantiation of dedicated BRAM buffers for the storage of event traces. However, the addition of such buffers and of the necessary communication structure (a bus shared amongst the counters) is quite straight-forward, inside the topology description file.

The overhead introduced while accessing the performance counters depends on three utilization factors, namely which core is going to access them, when they are accessed, how they are physically connected to the rest of the system. The allowed alternatives are:

- Regarding which core is intended to access the performance extraction subsystem, two main options exist. A dedicated processing core can be added to the emulated system, in order to perform the access to the event-counters without affecting the regular execution on the other emulated processors. Alternatively, in order to save hardware resources, the same processing cores of the emulated platform can interleave the execution of their instructions with the accesses to the performance counters.

- Concerning the time of the access to the performance counters, they can be accessed off-line (at the end of the execution) or at runtime, in case the read values should be used to implement runtime resources management mechanisms.

- Finally, the event-counters can be connected to the rest of the system via dedicated point-to-point connections, at the price of additional hardware resources to be utilized, or they can be accessed through the same interconnection layer already present in the emulated system, at the price of an overhead in the actual traffic pattern generated by the application.

Chapter 5

Methodologies for adaptive MPSoCs

The starting assumption of our system adaptivity approach, as depicted in the right part of Figure 5.1, is that we target an MPSoC composed of tiles, connected by a NoC, with completely distributed memories and no direct remote memory access. This means that the processing element of a tile can only directly access the content of its own local memory. All the communication and synchronization between processes mapped on different tiles can only happen using messages sent over the NoC.

Our approach for realizing system adaptivity consists of deploying the processes of the application(s) modeled as PPNs over the NoC-based MPSoC and allowing their run-time remapping to adapt the system to the changing operating conditions such as variation in quality of service requirements, availability of resources, or power budget constraints. In particular, system adaptivity in our system is supported by using a dedicated middleware, which is highlighted in the software infrastructure diagram in the left part of Fig. 5.1.

At the top of the software stack, applications are described by PPN processes implemented as separate threads. An example of a thread representing a PPN process is given in Fig. 5.3(b) and it will be described in detail in Section 5.1. However, in this work the basic structure of PPN processes has been adapted to ease the realization of a predictable process migration mechanism, as will be described in Section 5.2.

At the bottom of the software stack, the operating system (OS) is responsible for all kinds of process management (process creation, deletion, setting its priority, suspending or resuming it). These features are essential for the run-time management of the system, and in particular for the execution of process migrations. Moreover, each processor has multitasking capabilities thanks to the OS. In case of *many-to-one* mapping, i.e. when more than one process are mapped on the same processor, the scheduling is data-driven. This means that a process runs as long as it blocks in reading/writing from/to a FIFO buffer. When the process blocks, it yields the processor control to the next process in the ready queue in a round-robin fashion.

In between the applications and the operating system, we devised and implemented a middleware which comprises two main components. The first one is the *PPN communication API*, which realizes the communication and synchronization between processes located in separate tiles, according to the PPN semantics. The second one is the *Process migration API*, which deals with process creation/deletion, state migration and the other actions needed for run-time process re-mapping. The two middleware components will be

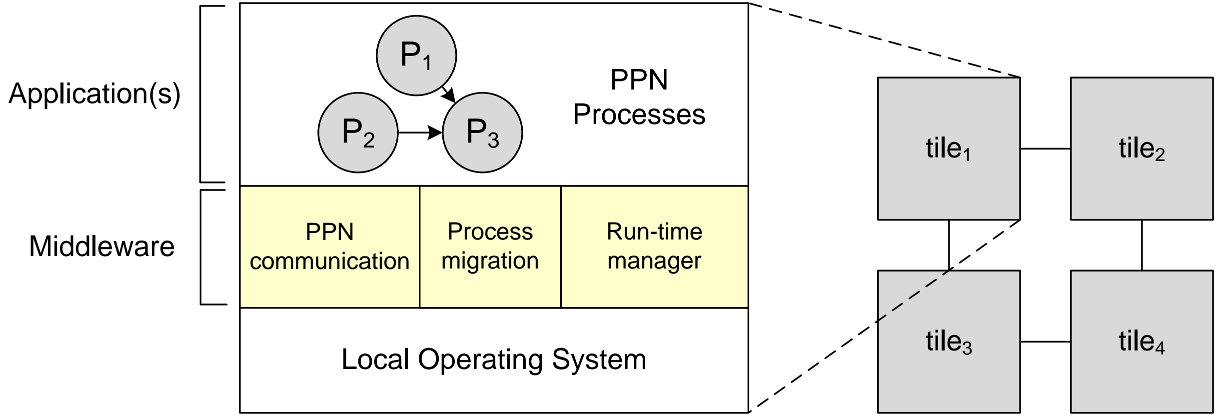

Figure 5.1: Software infrastructure for each tile of the NoC.

described thoroughly in Section 5.1 and Section 5.2, respectively.

5.1 PPN-over-NoC communication

This section describes the different solutions that we have devised and explored for the implementation of the PPN process communication and synchronization on a tiled NoC-based MPSoC. Basically, the devised approaches differ in the frequency of acknowledgment messages sent from a consumer process to a producer process about the status of the consumer FIFO buffers.

5.1.1 Some definitions

A PPN is a graph defined as a tuple $(\mathscr{P},\mathscr{C})$, where:

- $\mathscr{P} = \{P_1, ..., P_N\}$ is a set of processes;
- $\mathscr{C} = \{ch_1, ..., ch_K\}$ is a set of FIFO channels.

Each process $P \in \mathscr{P}$ has a set of input channels IC_P and output channels OC_P. The processes which write into IC_P are the *predecessors*, the processes which read from OC_P are the *successors*. The processing element (PE) onto which the process is mapped is denoted as $map(P)$.

For each channel $ch \in \mathscr{C}$:

- we can derive, using the pn compiler [46], a buffer size B which guarantees deadlock-free execution of the PPN;
- the producer process, which writes data to the channel, and the consumer process, which reads data from it, are denoted respectively as $prod(ch)$ and $cons(ch)$.

PPN processes communicate and synchronize using these FIFO channels. The PPN semantics forces a process to *block on read*, when trying to get a data token from an empty FIFO, and *block on write*, when trying to write data to a full FIFO.

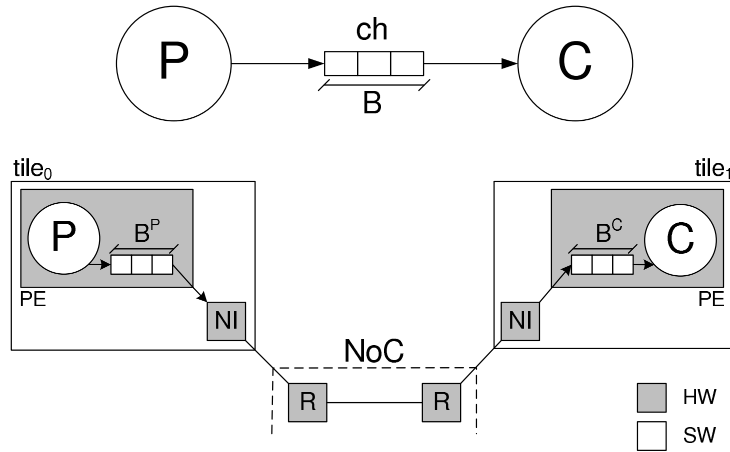

Figure 5.2: Producer-consumer pair with FIFO buffer split over two tiles.

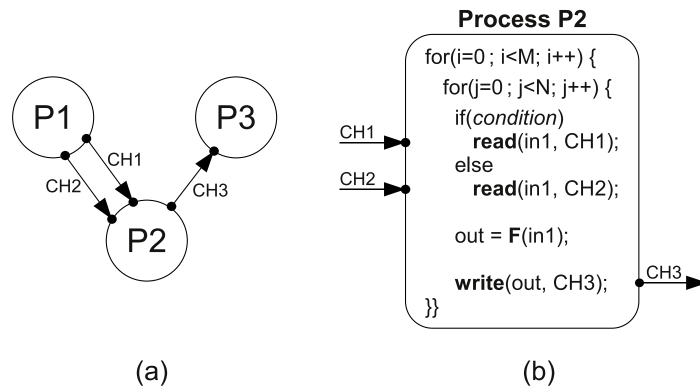

Figure 5.3: Example of a PPN (a) and structure of process $P2$ (b).

All PPN processes have the same code structure, an example of which is given in Fig. 5.3(b). Nested loops iterate, for a given number of times, the body of the process, which is split in three main parts. First, the process reads the input data tokens from (a subset of) the input channels. This is represented by the *READ* statements in the figure. Second, the process function (*F*) produces the output tokens by processing the input tokens. Finally, the output tokens are written to (a subset of) the output channels (*WRITE* statement).

The simplicity of the PPN process structure and semantics ease the development of system adaptivity support, as will be described further in the thesis. Only minor changes to the PPN process structure are needed to allow a predictable process migration mechanism, as will be described in Section 5.2.

5.1.2 Inter-tile synchronization problem

The main problem addressed in this section is the efficient implementation of a communication API allowing the execution of applications modeled as PPNs on Network-on-Chip MPSoC platforms. The first requirement is that this API must respect the PPN semantics. Moreover, we want our middleware to be application-independent and oriented to system adaptivity.

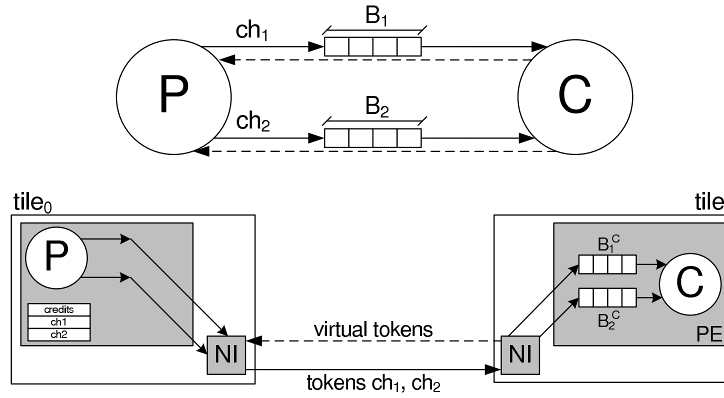

Figure 5.4: Producer-consumer pair using the virtual connector approach.

The communication and synchronization problem when mapping PPNs on a NoC is depicted in Fig. 5.2. Consider a producer P and a consumer C connected through an asynchronous communication FIFO buffer B. If both the producer and the consumer can directly access the status register of this FIFO buffer, to check whether it is empty or full, implementing the PPN semantics is straightforward. However, in NoC implementations with no direct remote memory access, processes can exchange tokens only via the network. Thus, we have to split the buffer B in B^P and B^C, one on the producer tile and one on the consumer tile. We want to implement the PPN semantics without a dedicated support from the underlying architecture that allows checking for the status of the remote queues. If $size(B)$ is the minimum buffer size that guarantees deadlock-free execution of the original PPN graph, the size of B^P and B^C must be set such that $size(B^P) + size(B^C) \geq size(B)$.

We do not require support for multiple hardware FIFOs on each NoC tile. The only hardware buffer of a tile resides in the Network Interface (NI). We just rely on the ability to transfer tokens, in both directions, from this buffer to the *software FIFOs* which implement the channels of our PPN.

Consider again Fig. 5.2. Even if the consumer process C can only access the status of B^C, implementing the *blocking read* is trivial because every time process C wants to access B^C and this buffer is empty, the consumer just has to wait until tokens arrive from the producer tile. However, since the producer process B can only access the status of B^P, implementing the *blocking on write* behavior is more difficult. The producer must know that the remote buffer B^C is not full before sending tokens to C over the NoC. There are several ways to notify the producer about the status of the buffer on the consumer side, and we will compare the approaches that we have investigated in the remainder of this section.

Furthermore, we want the communication API to take care of the distribution of processes among the NoC tiles with no influence on the application designer. This means that we want to maintain the code structure of the PPN application processes, an example of which is shown in Fig. 5.3(b). In particular, we want the communication primitives (*read*, *write*) of PPN processes to remain generic, without the notion of process mapping or platform details. These generic primitives are then translated by the communication API implementation in mapping- and platform- dependent function calls.

In all of the communication approaches described below, system adaptivity is taken into account by using dedicated middleware tables that list, among other information, the source and destination tile for each channel of the PPN graph. For instance, when a process is up

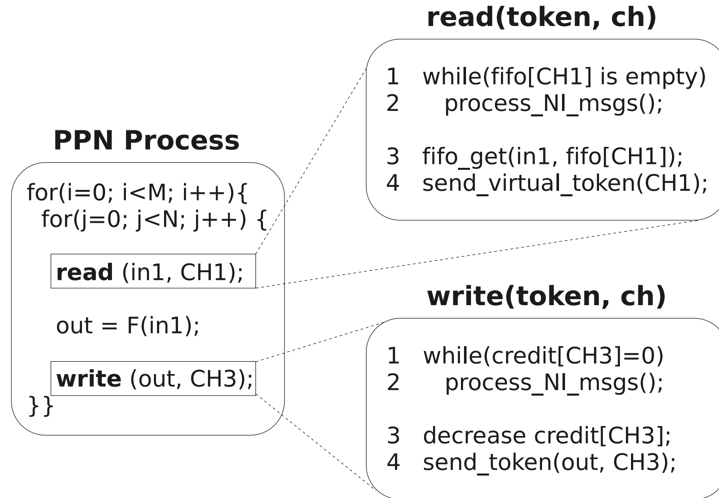

Figure 5.5: Pseudocode of the VC approach.

to send a packet to the consumer via a specific channel, the implementation of the *write* primitive will check in the middleware table what is the current destination of that channel. Then, it will place the packet in the NI output buffer, with the appropriate destination field of the header. As described in Section 5.2, these middleware tables are updated at run-time to allow run-time remapping of application processes over the tiles.

5.1.3 Virtual connector approach (VC)

In the virtual connector communication approach, which is depicted in Fig. 5.4, for every channel in the original PPN graph we add a virtual one in the opposite direction. This virtual connector is used for acknowledging the producer about the status of the FIFO buffer on the consumer tile. We adapted this approach, previously proposed in [13], to the needs of our system implementation. In that work the proposed communication middleware is *active*, meaning that it is implemented using separate threads which deal with the PPN communication, while in our implementation the middleware is *static*, with no separate threads for communication. Although a comparison of the static and active implementations may be worthwhile to do, for the moment we adopt the static approach with the argument that the scheduling and synchronization of additional middleware processes may introduce an additional overhead due to the context switching times.

For each channel in the original PPN graph we instantiate a software FIFO buffer on the consumer tile. The sizes of these buffers are set to the value of the original buffer size in the PPN graph. On the producer tile there are no software FIFOs when using this approach, because tokens can be directly sent over the network via the NI. This is due to the fact that the credit-system guarantees that enough locations are free on the remote buffers before sending a token. Therefore, referring back to Fig. 5.2, in this approach for each channel i, $size(B_i^C) = size(B_i)$ and $size(B_i^P) = 0$.

In our implementation, we store on the producer side a variable for each channel, called $credit$, which represents the number of free slots in the remote FIFO buffer implementing that channel. At startup, the credit is set to the size of the remote FIFO ($credit_i = size(B_i^C)$), because all of its slots are free. For each token sent over the network by the producer, the

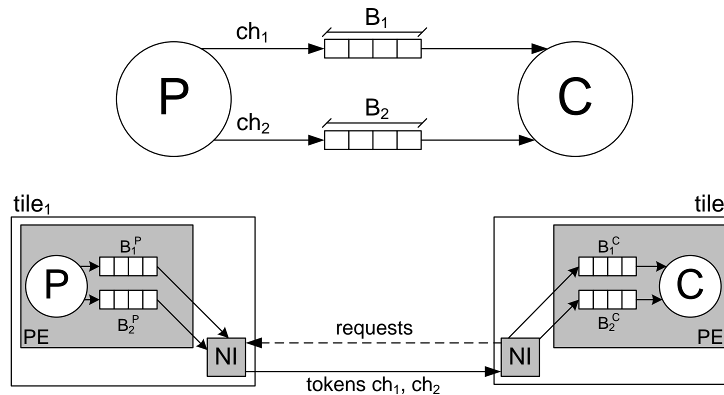

Figure 5.6: Producer-consumer implementation: when using the VRVC, the producer receives back virtual tokens (a); when using R, it receives requests (b).

credit of the corresponding channel is decreased by one. The producer is allowed to send tokens over the network only if the credit is positive, otherwise it blocks. This implements the *blocking write* behavior. On the consumer side, for every token consumed from that channel, a virtual token (VT) is sent back to the producer via the virtual connector. For every virtual token received on the producer tile, the credit of the corresponding channel is increased by one. In this way the producer is constantly updated about the status of the remote FIFO buffers.

The pseudocode of the VC approach is shown in Fig. 5.5. Both the *read* and *write* primitives use an auxiliary function, *process_NI_msgs()*, that is used when blocking on read or on write. This function checks the status of the NI buffer for incoming packets. If the buffer is not empty, it processes one packet at a time, until all the incoming packets are consumed, in the following way. If the packet is an incoming token for channel *i*, it stores the token in the software FIFO which implements channel *i*. If it is a virtual token for channel *j*, it consumes the packet and increase the credit of channel *j*.

In Fig. 5.5, lines 1-2 of the *read* primitive implement the blocking read. If the FIFO buffer corresponding to the calling channel (in the example, *CH1*) is empty, *process_NI_msgs()* is executed until new tokens for that channel reach the NI input buffer. Lines 3 and 4 complete the *read* primitive: the token is transferred from the software FIFO to *in1*, and a virtual token is sent back to the producer of *CH1*. This is actually done by putting in the NI outgoing buffer a packet representing a virtual token for channel *CH1*, as shown in Fig. 5.12.

Similarly, in the *write* primitive in Fig. 5.5, lines 1-2 implement the blocking write behavior. If the credit is zero, *process_NI_msgs()* is executed. If virtual tokens for the blocked channel are received, the credit is then increased and this condition unblocks the write to that channel. Lines 3-4 complete the *write* procedure. The credit for the considered channel is decreased, and the token is sent over the network, which is actually done by putting in the NI outgoing buffer a packet representing the token (refer again to Fig. 5.12).

5.1.4 Virtual connector with variable rate approach (VRVC)

This approach represents a variant of the *virtual connector* described above. The basic idea is that instead of sending one virtual token to the producer for *every* consumed token of channel i, the consumer sends it after n_i consumed tokens, where n_i is a parameter that

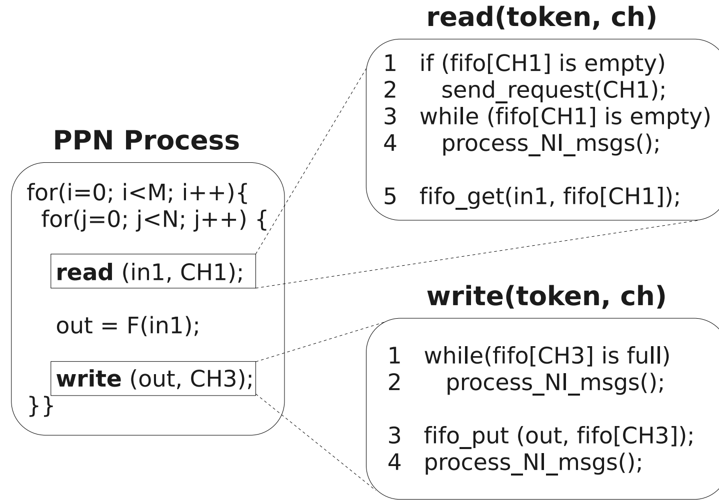

Figure 5.7: Pseudocode of the R approach.

can be set such that $\forall i \in \{1,\cdots,N_{ch}\}$ $1 \le n_i \le size(B_i)$, where N_{ch} represents the number of channels in the PPN graph. The *credit* variable for channel i will then be increased by n_i for every virtual token received for that channel. This approach leads to a reduced traffic on virtual connectors, which can be beneficial in NoC implementations to avoid congestion of packets.

Since the sending back of virtual tokens does not happen for every consumed token, in some cases the PPN graph properties require to store, also at the producer side, tokens for the channels in order to avoid deadlocks. This requires the adoption of software FIFO buffer also on the producer side. In the most generic case, the size of these buffers should be as large as the original buffer in the PPN graph. This means that $\forall i \in \{1,\cdots,N_{ch}\}$ $size(B_i^P) = size(B_i^C) = size(B_i)$, as depicted in Fig. 5.6, case (a).

5.1.5 Request-driven approach (R)

This method is very similar to the approach used in [30] for realizing the FIFO communication on the Cell BE platform. In this approach, the transfer of tokens from the producer tile to the consumer tile is *initiated by the consumer*. This means that every time the consumer is blocked on a read at a given FIFO channel, it sends a *request* to the producer to send new tokens for that channel. The producer, after receiving this request, sends *as many tokens* as it has in its software FIFO implementing that channel.

Since also in this case we need to store tokens both on the producer side and on the consumer side, we need software FIFO structures on both sides. The size of these buffers is set, for each channel i, to match the size of the queue in the original PPN graph (B_i), such that $\forall i \in \{1,\cdots,N_{ch}\}$ $size(B_i^P) = size(B_i^C) = size(B_i)$. This condition guarantees deadlock-free execution on the NoC and it is the same as in the *VRVC* approach. The structure of a producer-consumer pair using the *R* approach is shown in Fig. 5.6, case (b). Since the consumer buffer of a channel is empty when a request is made, and given that the FIFO buffers for that channel have the same size on both sides, there is always enough space to store tokens sent by the producer as a consequence of the request.

Fig. 5.7 shows the pseudocode of this communication approach. Similarly to the VC ap-

Table 5.1: Middleware table example

ch	prod(ch), cons(ch)	map(*prod(ch)*), map(*cons(ch)*)
1	P_1, P_2	$tile_0, tile_1$
2	P_2, P_3	$tile_1, tile_2$

proach, it makes use of the auxiliary function *process_NI_msgs()* to process incoming packets of tokens or requests. The main difference in this case is that this function is in charge of reacting to a received request message for a channel with the immediate sending of all the tokens contained in the software FIFO that implements that specific channel.

The *blocking on read* behavior is implemented in lines 1-4 of the read primitive in Fig. 5.7. When the software FIFO of the calling channel is empty, a request is sent to the producer tile of that channel, and the processor keeps executing *process_NI_msgs()* until a packet of tokens for the calling channel arrives. The *blocking on write* is implemented in lines 1-2 of the write primitive in Fig. 5.7. When the FIFO of the calling channel (in the example, *CH3*) is full, the processor keeps executing *process_NI_msgs()* until a request for that channel arrives.

5.2 Process migration

This section provides a description of the proposed PPN process migration mechanism over the MADNESS NoC-based MPSoC system. It is a fundamental part of the middleware depicted in Fig. 5.1 because it realizes the run-time re-mapping of processes, which in turn allows system adaptivity strategies.

The migration mechanism depends on the considered communication approach. As a starting assumption to devise the migration mechanism, we consider the *request-driven (R)* communication approach described in Section 5.1.5. This choice is made because the *R* approach leads to a considerably easier implementation of the migration mechanism since it requires less synchronization points. At the same time, it gives performance comparable to the other approaches for computation-dominant applications, as will be shown in Section 5.3.

We recall that to take into account the run-time remapping of processes over the NoC, each PE stores in its local memory a *middleware table* which is used to refine the generic communication primitives to mapping-dependent function calls. An example of a middleware table generated for the initial mapping in Fig. 5.8 is given in Table 5.1. For each channel of the PPN, the producer and consumer process IDs are stored, together with their current mapping in the system. Auxiliary information, for instance pending requests during migration execution, is also saved for each channel.

Mainly two kinds of process migration mechanism can be considered, namely *process replication* and *process recreation*. In process replication, the program code of a process that can be migrated is copied in each tile, thereby creating replicas of the process. When a process needs to be migrated from one tile to another, the process is suspended on the first tile and restarted on the second. The state of the process must be copied from the first tile to the second because the process cannot be just restarted from scratch.

The second kind of process migration mechanism is based on the so-called *process recreation*. In this case, if a migration is needed, the process is killed on the original tile it runs on and created on another tile by moving both the process code and state. The OS/middleware

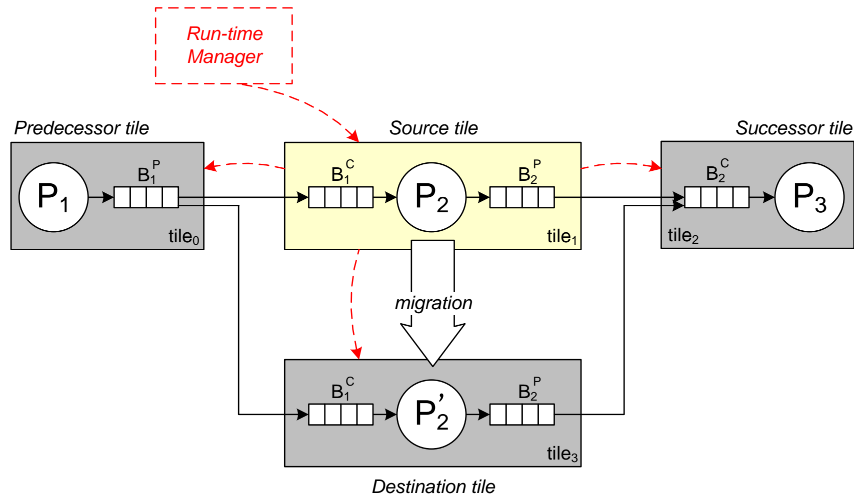

Figure 5.8: Migration diagram.

in this case must support dynamic loading of processes to processors. This way, only one instance of the process code exist at a given time in the system.

On the one hand, the process replication mechanism is less efficient in terms of memory usage, compared to the process recreation. On the other hand, it offers significant advantages such as easier implementation and faster migration procedure. We chose the process replication mechanism because we consider the fast execution of process migration more important. Moreover, the memory constraint in our system is not critical.

A simple diagram showing the migration of a PPN process is depicted in Fig. 5.8. Even though this is a simple example, it can be easily generalized for more complex PPN topologies. The diagram highlights the tiles involved in the process migration procedure, which are referred to as:

- the **source tile**, namely the tile which runs the process before the migration takes place;
- the **destination tile**, which is the tile that will execute the process after the migration;
- the **predecessor tile(s)**, which runs the predecessor process(es);
- the **successor tile(s)**, which executes the successor process(es).

The structure of PPN processes, modified to allow migration at any point during the execution of the process main bodies, and the proposed process migration mechanism are described in the following two subsections.

5.2.1 Migratable PPN process structure

Our goal is to allow the migration to be performed at any time during the execution of the process main body, in order to improve the migration response time. To this end, we extended the NI interface of a tile with the ability to generate an interrupt for the processing element when a message with a reserved tag is received, as described in Subsection 4.3.6. This extension has been made because the detection of migration decisions by polling at

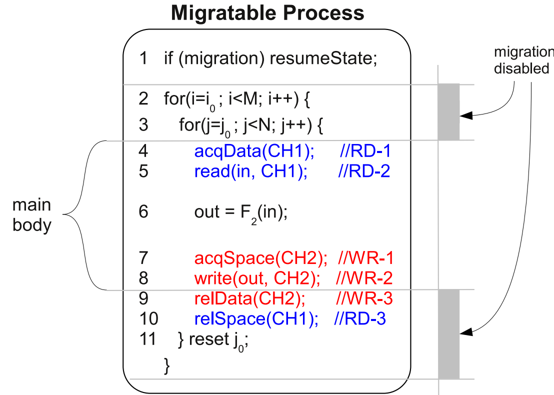

Figure 5.9: Migratable PPN process.

specific migration points in the code may cause undesired latency in the migration procedure.

With the requirement that migration may happen at any point within the execution of the processes main body, we devise the structure of a migratable PPN process as shown in Fig. 5.9. It is based on the structure shown in Fig. 5.3(b), which we will refer to as *basic process structure*.

We comment and motivate the migratable PPN process structure shown in Fig. 5.9 in the following. When the thread starts, in line 1, it checks if the *migration* flag is set. If the checking is positive, it means that a migration has been performed, so the process state is reloaded.

Since the PPN model definition requires a *stateless* process function, for example F_2 in Fig. 5.9, i.e., a function whose execution does not depend on the previous iterations, the state of a PPN process is represented only by:

- the content of its input and output FIFOs;
- its iterator set, namely the values of the nested loop iterator variables, see (i, j) in Fig. 5.9, lines 2-3;

When a function requires to have a state, it is represented in the PPN model by a stateless function with FIFO self-edges, which represent the function state.

Both state components listed above are transferred from the source tile to the destination tile upon migration. If the migration flag is false, it means that the process starts from scratch, with empty input and output FIFOs and $i_0 = j_0 = 0$.

Lines 2 and 3 differ from the basic process structure in Fig. 5.3(b) because the iterators inside the *for* loops do not start from zero in case of migration. Instead, they start from the values i_0 and j_0, which represent the iteration at which the process was interrupted by the migration while running on the source tile. After the first complete execution of the inner *for* loop, starting from j_0, the value of j_0 is set to zero in line 11 such that the next execution of the inner loop starts correctly with $j = 0$.

The communication primitives are different from the ones used in the basic process structure. The *read* primitive, for instance, is split into three separate operations (see lines 4, 5, 10). First, the input channel (*CH1*) is tested to verify the presence of an available data token, using the *acquireData* function (*acqData(CH1)* in line 4). Then, the token is actually

copied from the software FIFO to the input variable which will be processed by the process function F_2. The copy operation is performed in line 5. However, differently from the normal *read* primitive, the memory locations occupied by the read token are not released immediately. The actual release, which consumes the data from the FIFO by increasing the read pointer, takes place only in line 10 (*relSpace(CH1)*). This way, if a migration is triggered before the release instruction, the process can be correctly resumed on the destination tile since it will read again the same input token, because the read pointer is not changed. Similarly, the *write* primitive is split in three operations, see lines 7, 8, 9, of which only *relData* affects the write pointer. Finalizing the *read* and *write* operations at the end of an iteration allows the process migration to happen anywhere within lines 4-8 correctly. Note that, in case of multiple input or output channels, the release operations should be grouped together and placed right after the main body of the process, in order to guarantee a consistent process state.

Process migration cannot happen within the lines 9-11 and 2-3 because that would cause an inconsistency in the migrated process state. This is because lines 9 and 10 can be considered as an update of the output and input FIFOs state, while lines 11, 2 and 3 represent the iterator set update. If, for instance, a migration happens after the FIFO state update but before the iterator set update, the migrated process will re-start the execution with the FIFO status corresponding to the next iteration, but with the iterator set of the current (interrupted) iteration. This condition will certainly cause a deadlock. Although the process migration cannot happen within lines 2-3 and 9-11, we note that these sections represent a minimal part of the process execution, because performing the update of read and write pointers and iterator sets is a matter of a few simple instructions. Therefore, disabling the migration within these sections does not increase the response time significantly.

The principle behind the proposed migratable process structure is that the state of a process must be *consistent* and *up-to-date* when a migration is performed. This allows the migrated process to correctly resume its execution on the destination tile. Leveraging the PPN process structure, our approach does not require the designer to specify the context that has to be transferred upon migration as in [7]. This burden is neither moved to the OS/middleware level as in [3]. Determining the state to be migrated is not needed because the PPN process state simply consist of the two components described above. Moreover, our approach does not need designer-generated checkpoints/migration points. The resource manager in Fig. 5.8 can interrupt the process execution at any time during the execution of the process main body. The migrated process will then resume its execution from the beginning of the interrupted iteration. On the one hand, this implies that if the migration is triggered in the middle of the function execution, the time since the start of the iteration is lost. On the other hand, this approach leads to a more efficient implementation and predictable migration response time, which we consider more important for our goals.

5.2.2 Process migration mechanism

The migration mechanism requires actions from all the tiles depicted in Fig. 5.8. The migration decision is taken by the resource manager, which sends a specific control message to the source tile. How the resource manager takes the migration decision will be described in further detail in Chapter 6. The source tile then broadcasts this control message to the destination, predecessor and successor tiles to complete the migration procedure.

The control messages which notify the process migration to the involved tiles contains the ID of the migrated process (*ctrl_msg.migProc_ID*) and the new mapping of that process (*ctrl_msg.dest_PE*). On all of the involved tiles, and on the resource manager, the middleware tables are then updated by executing the following operations, for each channel in the list:

- if (*prod(ch)==ctrl_msg.migProc_ID*)
  update *map(prod(ch))* to *ctrl_msg.dest_PE*
- if (*cons(ch)==ctrl_msg.migProc_ID*)
  update *map(cons(ch))* to *ctrl_msg.dest_PE*

For each of the tiles involved in the migration procedure, the detailed list of required actions are explained below.

Actions on the source tile

On the source tile, the process has to be stopped, and its state saved and forwarded to the destination tile. Moreover, the middleware table is updated as described above. The source tile takes also care of propagating the migration decision to the other tiles involved in the migration procedure. This propagation is depicted by the dashed arrows in Fig. 5.8.

Actions on the destination tile

The destination tile receives a specific message for process activation. The migration procedure is handled by creating the required software FIFOs and by activating the replica of the migrated process using the corresponding OS call. Before the process replica is started, the *migration* flag is set to 1 so that the state of the migrated process is resumed (see line 1 in Fig. 5.9). This implies that the input and output FIFOs of the migrated process are copied, and the iterator set (in the figure, i_0 and j_0) are set such that the execution starts from where it was suspended on the source tile. The middleware table is also updated in the way described above.

Actions on predecessor tile(s)

On these tiles, the only required step is the update of the middleware tables according to the new mapping of the migrated process. This way, new tokens meant for the migrated PPN process will be sent to the destination tile.

A corner case of the communication between the migrated process and its predecessors may happen when the process has sent a *request* for new tokens just before the migration command arrives. If that request has been served, it means that new tokens are either traversing the NoC or they are already stored in the source tile. The predecessor tile in this case has to send another interrupt-generating message to the source tile, in order to force the forwarding of these data tokens to the destination tile.

Actions on successor tile(s)

Similarly, the successor tiles have to update the middleware tables so that the new requests for data tokens will be sent to the destination tile. A particular case in the protocol between successor processes and the migrated process is represented by requests which are sent to

Figure 5.10: PPN specification of the Sobel filter.

Table 5.2: Execution times of Sobel functions

Process	Execution time (c.c.)
readPixel	5
gradientX	31
gradientY	31
absValue	118
writePixel	5

the source tile just before the interrupt decision takes place. In this case, if the requests are not served before the migration, they have to be forwarded to the destination tile.

5.3 Experiments and results

In order to evaluate the proposed middleware, we perform two experiments to assess both its main components. In the first experiment, described in Section 5.3.2, we compare the efficiency of the different approaches for the PPN communication API in two case studies. In the second experiment, described in Section 5.3.3, we assess the process migration benefits and overhead by applying our migration mechanism in one of the case studies. Before presenting these two experiments, we describe the case studies and the experimental setup that we used to obtain the results.

5.3.1 Case studies and MPSoC platform setup

We evaluate the three communication approaches presented in Section 5.1 on two applications modeled as PPNs with extremely different communication/computation characteristics. The reason is that we want to compare the overhead of the different approaches between two extremes. The Sobel filter application described in Section 5.3.1 represents the worst case (the first extreme), when the computation/communication ratio is low and the PPN topology is complicated. The M-JPEG encoder application described in Section 5.3.1, on the other extreme, is computation dominant and with relatively simple PPN topology, therefore represents the best case. We describe briefly the two case studies in order to allow a better understanding of the obtained results. We also provide an overview of the generated platform instance that we use to run the experiments.

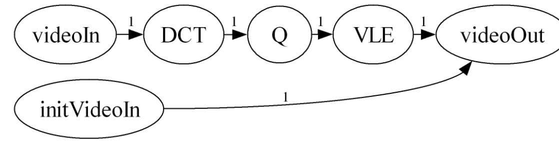

Figure 5.11: PPN specification of the M-JPEG encoder.

Table 5.3: Execution times of M-JPEG functions

Process	Execution time (c.c.)
initVideoIn	18
videoIn	1910
DCT	126386
Q	69238 (avg)
VLE	46688 (avg)
videoOut	1292 (avg)

Sobel filter

The Sobel application is an edge-detection algorithm for digital images. Its PPN graph is shown in Fig. 5.10, where the numbers over edges indicate the minimal buffer sizes needed for processing a 200x122 pixel input image. The PPN processes which comprise this application are very lightweight in terms of computation. The numbers of clock cycles required for one execution of each function are summarized in Table 5.2. For all of the channels in the graph, the size of exchanged tokens is 4 bytes, and the number of written tokens is 23760. From these metrics it is clear that the Sobel application is largely communication-dominant.

M-JPEG encoder

The PPN specification of this application is shown in Fig. 5.11. The size of tokens ranges between 16 and 1024 bytes, and all of the channels are written 128 times, except the output of *initVideoIn* which is written only once. The numbers of clock cycles required for the execution of each function of the M-JPEG application are summarized in Table 5.3. This application shows a much simpler communication and synchronization pattern compared to Sobel, and it also has a much higher computation/communication ratio.

MPSoC platform setup

The system on which we evaluated our communication approaches is based on a 2x2 mesh of tiles with NoC interconnection, generated through the MADNESS framework. Each tile is composed by a MicroBlaze processor, with its local program and data memories, and a Network Interface.

The Network Interface contains only two hardware FIFOs, one for packets which are incoming from the NoC, and one for packets that have to be injected in the NoC. The processor is able to quickly access the status of the incoming hardware FIFO, via a dedicated signal, to see if there are messages to be forwarded from the NI buffer to the SW FIFO buffers that implement channels of the PPN graph. In the opposite direction, when a packet has to be sent

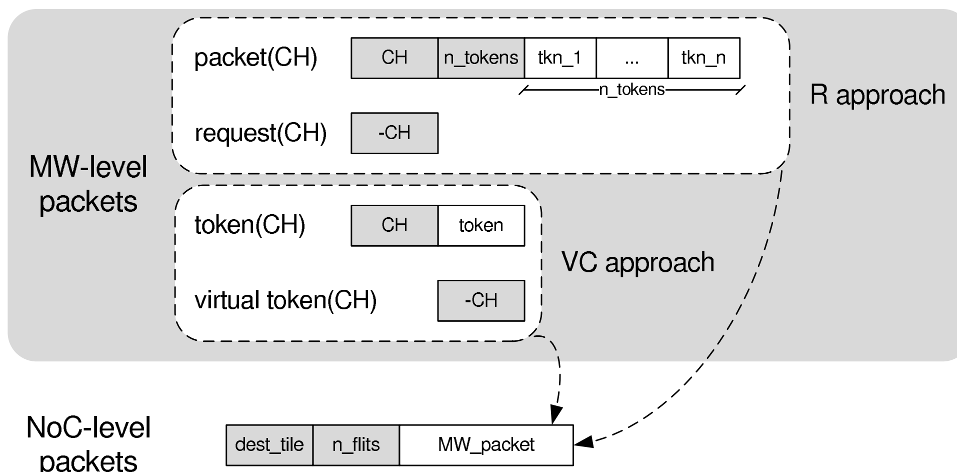

Figure 5.12: Structure of middleware- and network- level packets.

over the NoC, the processor forwards data from its local data memory to the outgoing NI hardware FIFO. Then the NI injects the packet in the network with the appropriate header (destination tile and payload size fields). The packets, as already described in Chapter. 4, are sent over the NoC using wormhole routing.

The actual structure of the different kind of messages that are sent over the NoC is represented in Fig. 5.12 for the *VC* and *R* communication approaches. At NoC-level, the packet comprises a NoC header that indicates the destination tile and the size of the payload, and the payload itself, which is the middleware-level packet (denoted as MW-level packets in the figure). The structure of middleware-level packets depends on the communication approach. In the *R* approach, a request for channel number i is implemented as a single flit, with value $-i$. By contrast, a packet used for transferring tokens has a header composed of two flits (channel number, number of sent tokens) and a payload with the sent tokens. The field that indicates the number of sent tokens (*n_tokens*) is necessary because this number is determined at run-time, when a request for that channel is received. The structure of middleware-level packets in *VC* is very similar, the only difference is that there is no need for a *n_tokens* field because in this method there is no packetization of tokens, i.e., *n_tokens* is always equal to one.

5.3.2 Inter-tile communication efficiency

The MPSoC generated platform described in Section 5.3.1 has been implemented on a Virtex5 FPGA prototyping board. We run the two application case studies using all the communication approaches proposed in Section 5.1 to obtain the results described below. The experiments for process migration are also described later in this section. Note that the proposed migration mechanism is generic, meaning that it is not dependent on a particular NoC implementation.

In order to compare the efficiency of the inter-tile communication of the different communication approaches, we execute the two case study applications with fixed mappings shown in Fig. 5.13. We chose these mappings because they expose the maximum amount of inter-tile communication. Therefore, the obtained results are largely dependent on the efficiency of the communication approach.

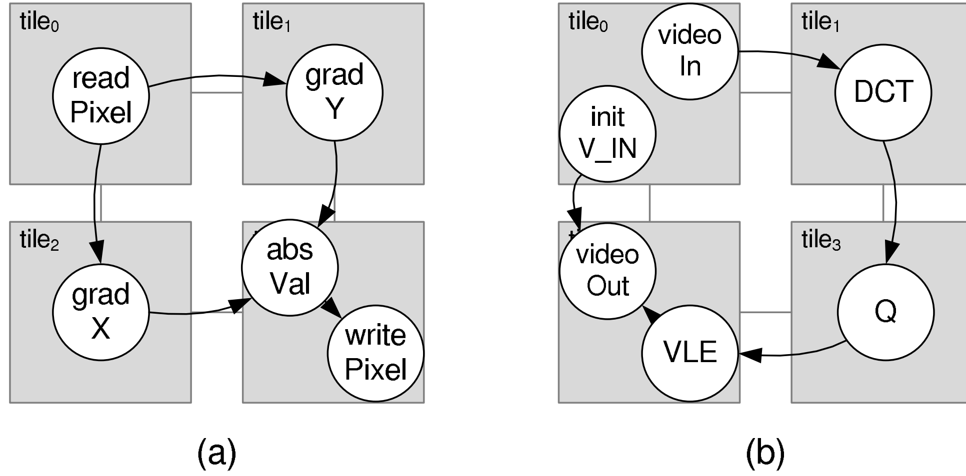

Figure 5.13: Fixed mappings for Sobel (a) and M-JPEG (b) to test the different communication approaches.

We found out experimentally that the parameter n_i of the VRVC approach gives the best performance when it is set to its maximum value, i.e. when $\forall i \in \{1, \cdots, N_{ch}\}\ n_i = size(B_i^C)$. The performance results, summarized in Fig. 5.14, show a large difference of the execution time for the Sobel application when using different communication approaches. However, in the M-JPEG case all of the communication approaches yield to similar performance results. The VC approach performs much better, compared to the others, in the Sobel application because its implementation does not require storing of tokens on the producer tile. This leads to a faster communication process, because it avoids the double copy (output variable → software FIFO → NI buffer) that is necessary in the other cases. We argue that the obtained results may change for NoC platforms with Direct Memory Access (DMA) cores that can benefit more from the packetization of tokens allowed in the VRVC and R approaches.

In order to evaluate the overhead imposed by the use of the NoC interconnection and our communication approaches, we implemented customized point-to-point systems, for both applications, as a baseline reference. In point-to-point systems, generated using the ESPAM tool [33], a dedicated hardware FIFO is instantiated for each channel of the PPN graph. In this way, the hardware platform perfectly matches the PPN MoC semantics. Obviously, customized point-to-point implementations do not allow for system adaptivity because all the design decisions (e.g., process mapping) have to be made at design time. It is clear that in our NoC system we sacrifice performance (especially for communication intensive applications) for adaptivity, the ability of managing the system at run-time, and generality, since the system is able to execute any kind of application modeled as PPN. The performance slowdown, when comparing the NoC system with the point-to-point systems, is shown in Fig. 5.15. It is noticeable that the Sobel application is highly penalized in the execution on our NoC system, whereas the M-JPEG application performs well because of its higher computation/communication ratio and its regular communication pattern.

The reasons why the communication onto the NoC platform is less efficient are mainly twofold. The first reason is that in this implementation, several PPN channels have to share the same physical channel (the NoC link). The second reason is a consequence of the first one. In the NoC case, the presence of only one physical link, being shared between different PPN channels, poses the need for a flow-control policy. To optimize for low hardware

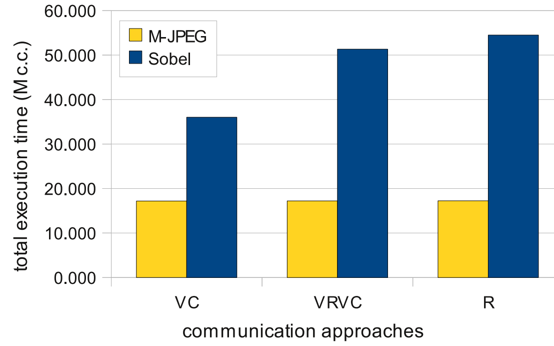

Figure 5.14: Total execution time for different communication approaches.

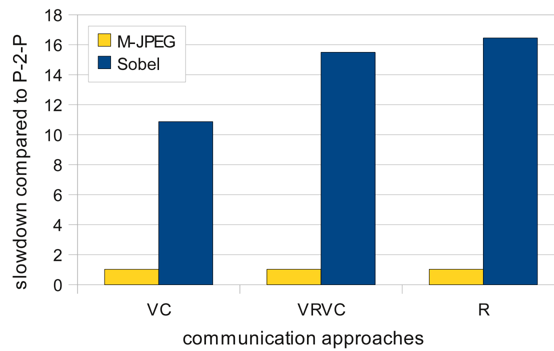

Figure 5.15: Slowdown for different communication approaches.

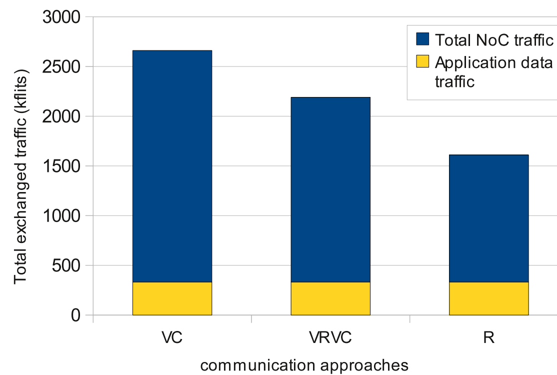

Figure 5.16: Traffic injected into the NoC by executing Sobel with different communication approaches.

overhead, we chose to implement the control flow at the middleware level, based on software FIFOs on the producer and on the consumer side. This requires additional memory copy operations to dispatch/multiplex the communication tokens to/from the correct software FIFO. Such copies are unnecessary in the case of adoption of multiple point-to-point connections with hardware FIFOs.

Another important metric when executing applications on a NoC-based MPSoC is the amount of generated control traffic overhead. In the *VC* case, for instance, this overhead is

represented by the NoC-level and MW-level headers, together with all the traffic generated by the virtual tokens. Ideally, the middleware should be designed to generate as less control traffic overhead as possible.

Focusing on the Sobel application, since it has the most complex communication pattern, we profiled the amount of traffic injected in the network, depending on the communication approach that is used. The results, depicted in Fig. 5.16, show two extremes: the *VC* and *R* approaches. This large difference can be explained by two factors. The first factor is the overhead of packet headers. On the one hand, in the *VC* approach, since there is no packetization of tokens, each token travels in the NoC with its own header. On the other hand, in the *R* approach, the producer sends as many token as present in its software FIFO in the same packet and therefore with the same header. The second factor is that the traffic on virtual channels in *VC* is much more than the traffic generated by requests in *R*. This is because in the *VC* approach a virtual token is sent back to the producer for every consumed token, whereas in the *R* approach the requests are made less frequently, just when the consumer is blocked on reading.

5.3.3 Process migration benefits and overhead

System adaptivity requires the ability to change the process mapping at runtime in a predictable and efficient way. To illustrate the benefits of our migration approach presented in Section 5.2, we compare our proposed migration mechanism, driven by interrupt-generating control messages, with a migration approach based on migration points.

In the latter case, process migration can take place only at fixed points in the code. The setup of this experiment is shown in the left part of Fig. 5.18. We use as a case study the M-JPEG application described in Section 5.3.1. $Tile_1$ initially runs all of M-JPEG processes, which are listed in Fig. 5.19. P_1 is derived by merging *initVideoIn* and *videoIn* processes, P_2 and P_3 represent respectively the *DCT* and *Q* processes, and P_4 is obtained by merging the *VLE* and *videoOut* processes. We use the M-JPEG application as a case study because, compared to the Sobel application, in M-JPEG processes are coarse-grained with high computation/communication ratio and therefore M-JPEG represents better the kind of applications which are likely to be mapped on a NoC-based MPSoC. The scheduling of the M-JPEG processes on $Tile_1$ before the migration is represented in Fig. 5.17. Scheduling charts have been obtained using the GRASP [19] trace visualization tool to plot the information gathered at run-time. The trace shows the periodic scheduling which is executed when all the processes are mapped on one tile and the scheduling policy is data driven. The buffer size of all the FIFO channels is set to two in this experiment. In this scenario, the process scheduling iterates in the following way. First, P_1 executes two times, until it blocks on writing because its output buffer is full. Then P_2 is scheduled. It completes two iterations, consuming the tokens created by P_1 and producing two tokens for P_3. It then blocks while reading its input FIFO which is empty by then. Similarly, P_3 and P_4 execute twice before blocking on read. This scheduling repeats until the end of the application execution if no migration is performed.

In Fig. 5.17, the arrows over the bars of process P_1 represent the start of an iteration of that process (for the sake of clarity, see line 4 in Fig. 5.9). Assume that these points correspond to migration points, namely where the process checks if migration-messages have been sent by the resource manager. Given that the migration request can reach $Tile_1$ at any time, the latency of the actual process migration can vary. In the best case, the migration re-

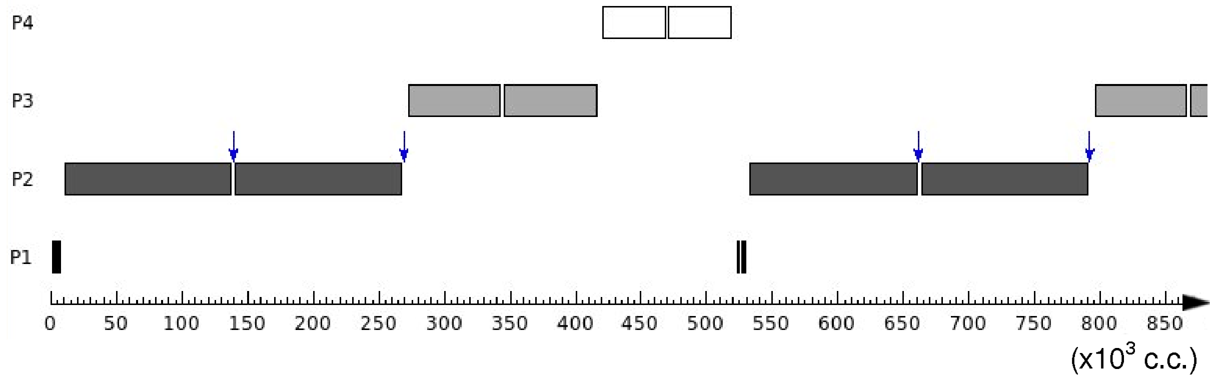

Figure 5.17: M-JPEG process scheduling when running on a single tile.

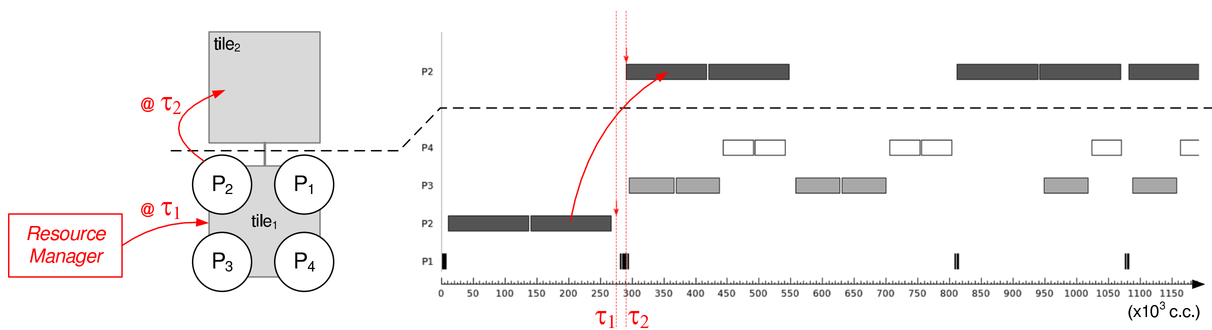

Figure 5.18: M-JPEG process scheduling while migrating P_2 using the proposed migration mechanism.

quest reaches the tile right before a migration point. In the worst case, the migration request arrives just after a migration point, for instance the one which is reached around clock cycle 275,000. The actual migration would not take place until the next migration point, which happens to be after 2 executions of P_3, P_4 and P_1, and one execution of P_2. In this simple case, an upper bound of the process migration response time can be found, based on the process scheduling, which in turn depends on the workload of processes, the buffer sizes and the scheduling policy. In more complex cases, where the scheduling on one tile is affected by the scheduling on other tiles because of data dependencies, even finding an upper bound for the response time practically would not be possible.

By contrast, the interrupt-driven migration mechanism that we propose in Section 5.2 has a predictable behavior. As shown in Fig. 5.18, the system has a faster response time to migration requests. At time τ_1, which is the worst case for the fixed point migration strategy discussed above, the resource manager sends a control message which triggers the migration of P_2 to $Tile_2$. The process can be restarted on the destination tile within a predictable amount of time represented by the difference $(\tau_1 - \tau_2)$. This is the time it takes the source tile and the destination tile to execute the steps described in Section 5.2, such as the movement of the process state and the activation of process P_2 on the destination tile. This migration overhead in time $(\tau_1 - \tau_2)$, as shown in Fig. 5.18, is way smaller than a single execution of the DCT function in process P_2. The migration procedure in this example actually takes less than 12% of a single execution of the DCT process.

Note that an upper bound of the migration procedure overhead can be derived for guaranteed throughput (GT) NoCs. In fact, the migration duration T_{mig} of a process $P \in \mathscr{P}$ can be split in two main components:

$$T_{mig}(P) = T_{stateMig}(stateSize(P)) + T_{procAct} \quad (5.1)$$

$T_{procAct}$ is a constant value which represents the time required to activate the migrated process using OS system calls, to update the middleware table, and complete all the actions described in Section 5.2 on the destination tile. $T_{stateMig}$ is the time it takes to transfer the state from the source to the destination tile. Its worst case, for GT NoCs, depends only on the state size. The largest state size of a process P is obtained when both the input and output FIFO buffers of P are full. This worst-case value can then be derived from the PPN topology and buffer sizes:

$$\max(stateSize(P)) = \sum_{ch \in IOC_P} size(B(ch)) \quad (5.2)$$

where $IOC_P = IC_P \cup OC_P$ as defined in Section 5.1.1, $size(B(ch))$ is the size of the buffer which represents the channel ch on the source tile. The value $size(B(ch))$ is obtained by multiplying the number of tokens of $B(ch)$ by the token size of a channel ch. An upper bound of the migration time T_{mig} of a process P can be calculated using $\max(stateSize(P))$ in Equation 5.1.

The worst case, for our interrupt-driven migration mechanism, is represented by the arrival of a migration request just before the end of a function execution in a process that has to be migrated. In this case, the migration still takes place in a predictable amount of time but the process execution has to roll back to the beginning of the interrupted iteration. All the time spent in the function execution is wasted in this scenario.

The proposed process migration mechanism allows our system to change its configuration at run-time. The resource manager triggers process migrations such that the system dynamically moves from the configuration (a) to (b), then to (c) in Fig. 5.19. By doing this, the resource manager is capable of changing, at run-time, the total execution time (T_{exe} in Fig. 5.19) and total exchanged traffic over the NoC (*NoC traffic* in Fig. 5.19). Both T_{exe} and *NoC traffic* correspond to the processing of a single input frame using the M-JPEG application. The resource manager, for instance, can decide to change system configuration because of a quality of service requirement demanded at run-time by the user.

In detail, in Fig. 5.19(a), all the processes of the M-JPEG application are executed on one tile, and the communication between processes does not happen via the NoC. In this configuration, the execution time per one frame is $T_{exe} = 33.073$ millions of clock cycles. However, in Fig. 5.19(b) processes P_1, P_3, P_4 are executed on one tile and process P_2 runs on a separate tile. Since process P_2 (the DCT) is the most computationally intensive in M-JPEG, accounting for 51% of the total workload, the obtained speedup compared to (a) is close to 2. In fact the execution time per frame drops to $T_{exe} = 17.342$ millions of clock cycles. The mapping in Fig. 5.19(c) does not show a relevant further performance improvement because P_2 represents the bottleneck of the M-JPEG application, such that even just migrating it to a separate tile as in (b) gives almost optimal performance. This experiment shows the efficiency of the proposed process migration procedure. The system is allowed to substantially change its execution time per frame, at run-time, with an almost negligible overhead. As explained above, the process migration overhead is way smaller than a *single* execution of the DCT

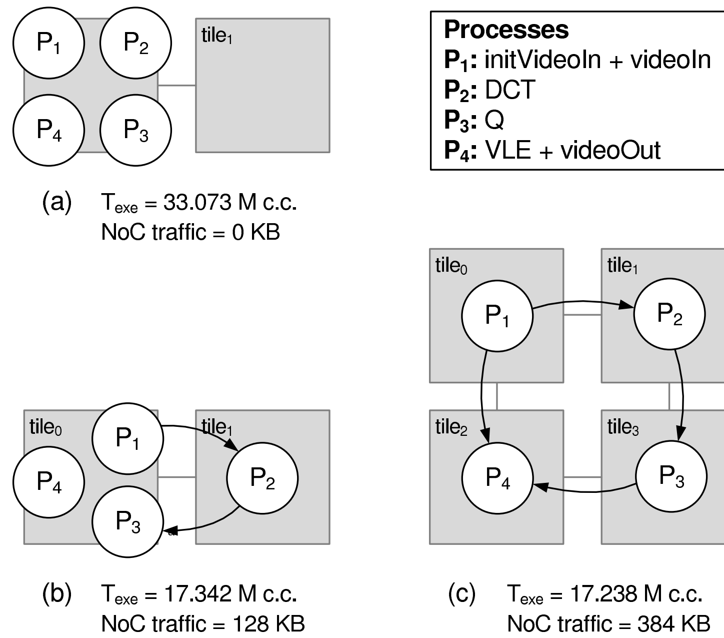

Figure 5.19: Execution time and generated traffic as a function of the process mapping. Only inter-tile communication links are depicted.

process. The negligible performance speedup obtained when changing the system configuration from (b) to (c) does not depend on the migration overhead. It is actually caused only by the intrinsic structure of the M-JPEG application.

Chapter 6

Fault-tolerance support within the MADNESS framework

6.1 Proposed approach

The MADNESS project focuses on the development of fault-tolerance solutions which are not dependent on a technology-related low-level fault model, but rather on technology-abstracting functional-level error models. The implemented fault-tolerance approaches focus on the detection of run-time faults and on the use of reconfiguration strategies at different levels. In the MADNESS platform, three main types of components are considered, i.e., *processing cores*, *storage elements*, and *NoC modules*. In this thesis, the solutions proposed for the case of processing cores are described.

6.1.1 Fault detection

For the detection of faults in the processing cores, one of the two approaches are used depending on the criticality of the application.

Self-testing module

If the application is not critical and a limited amount of error propagation is acceptable, a self-testing routine is executed periodically by the processing element to detect its permanent faults [38]. The self-testing module (shown in grey in Figure 4.3) calculates a signature of the results of the execution of the software routine, and compares it with a pre-calculated and pre-loaded correct signature. In case of a mismatch, a *fault* detection signal is raised. The self-testing routine should have a high fault coverage, a small code size and a fast execution time.

PPN-level self-checking patterns

For critical applications, concurrent self-checking techniques are employed at the process network level [13]. In the case of the *N-modular redundancy* (NMR) pattern, *N* instances of the same task are created and guarded within a *fork* and a *voter* task. The fork task simply forwards same copies of the token to each redundant instance of the task, whereas the voter

task determines the most recurring result produced by the redundant task instances. For $N \geq 3$, the voter is able to detect the faulty node and mask the error. In order to yield higher reliability, the redundant instances should be mapped onto different processing elements. The task graph can be transformed with patterns in various ways leading to different levels of reliability.

6.1.2 Task migration hardware module

Task migration support, described in detail in Chapter 5 can be used as a reconfiguration mechanism to survive in presence of faulty processing cores. However one fundamental restriction in such a scenario is that the faulty processor cannot aid in carrying out the migration procedure. As a remedy to this problem, a task migration hardware (TMH) module is proposed which is responsible for extracting the critical data from the faulty tile.

As shown in Figure 4.3, the TMH resides alongside the network adapter of each tile. It receives a fault detection signal from the self-testing module. Upon the detection of the fault, the TMH initiates the migration procedure that consists of the following steps:

1. the TMH isolates the faulty processing core,
2. the TMH notifies the run-time manager (RM) that resides on the fault-free core with the nearest bigger index,
3. the RM calculates the new mapping of the tasks according to the remapping heuristic,
4. the RM informs the predecessor, successor and all other tiles for the new mapping of each task on the faulty node,
5. the predecessor and successor tiles send a flush message to the faulty node,
6. the TMH receives the flush messages from all predecessor and successor tiles,
7. the TMH sends the state of all tasks and channels (pending requests and FIFO tokens) to the RM,
8. the RM receives the state and tokens of the tasks on the faulty node,
9. the RM carries out the software-based task migration (without updating the predecessors and successors again) as described in Section 5.2.

In step 6, TMH waits for all flush messages which guarantees that the tokens (from the predecessor tiles) and the requests (from the successor tiles), which may be in transit on the NoC at the time of fault detection, are received at the faulty node before TMH sends the migration data to the RM.

The TMH and the software-based task migration procedure are loosely coupled such that the modifications to the software-based task migration procedure in the later stages would not affect the functionality of the TMH, thus incurring minimal changes to the TMH, if any.

The TMH module carrying out this functionality (except step 6) has been designed and integrated into the MADNESS platform. The main figure of merit adopted when designing this module has been circuit complexity, so as to guarantee that failure rate will be much lower than the processing core.

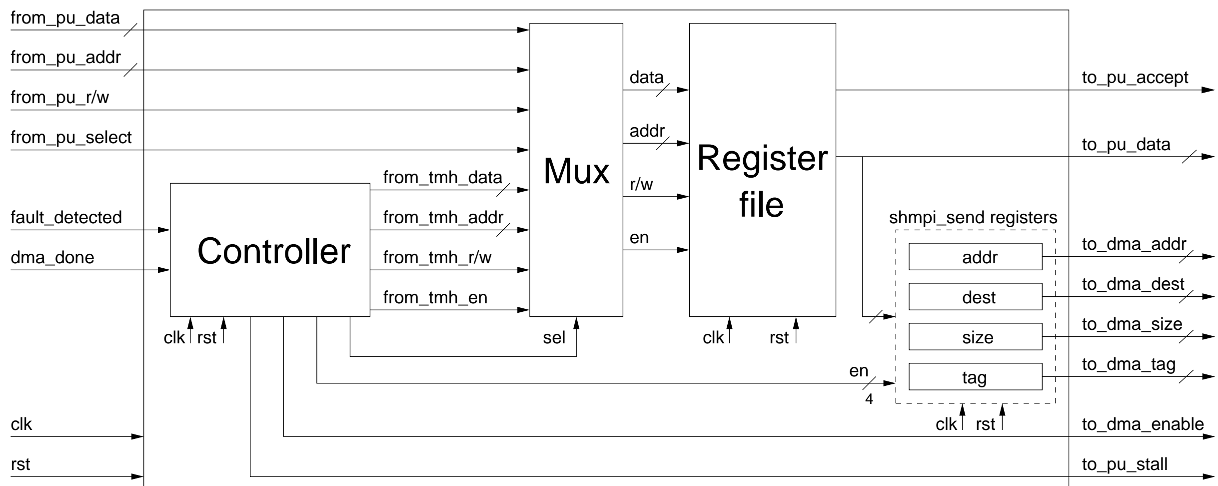

Figure 6.1: Interface and internal block diagram of the task migration hardware module

The interface and the internal block diagram of the TMH is shown in Figure 6.1. The interface consists of ports allowing:

i. to receive the fault detection signal from the self-testing module (*fault_detected*),

ii. to isolate the processor (*to_pu_stall*),

iii. to be read/written by the processing element from/to the register file inside the TMH (*from/to_pu_**),

iv. to send data via the NoC (*to_dma_**).

The TMH consists of a control unit implementing the finite state machine, a register file, a multiplexer and shmpi_send registers. The register file contains memory-mapped registers which store:

i. a pointer to the fault detection control message stored statically in the main memory,

ii. the tile ID that acts as the RM for the tile,

iii. the size of the control message,

iv. the special tag value used to send data carrying interrupt messages over the NoC,

v. the tasks mapped on the tile,

vi. the pointer to the array storing task states,

vii. the size of the task state,

viii. the special tag value used to send task states to the RM,

ix. the channels mapped on the tile,

x. the special tag value used to send channel data to the RM,

xi. a reduced middleware table containing for each channel the pointer to the software FIFO, the number of tokens in the channel, the size of the token type and a pending request flag.

When an application is launched, the PE initializes the TMH registers. During normal execution (when the PE is not faulty), whenever there is a read or a write, the number of tokens is updated in the TMH register for the corresponding channel. The read and write operations of the TMH take only cycle in order to reduce the overhead of the update operation. After the fault detection, TMH carries out a number of shmpi_send operations using the programmable DMA to notify the RM, to send states of mapped tasks and the data of mapped channels.

6.1.3 Online task remapping strategies

A fundamental step in the fault-tolerance support is determining the new cores where the tasks formerly executed by the faulty cores shall continue their execution. In order to provide a graceful degradation, the remaining fault-free cores of the platform should be used as optimally as possible. The remapping problem can be solved by an exhaustive analysis done at design-time that evaluates all possible fault scenarios of the system and embeds in the memory the optimal remapping results to be used when faults are encountered. An alternative approach is using online task remapping heuristics, whereby the decision is taken by a remapping heuristic executed at run-time. Such an approach requires less memory, does not require a heavy design time analysis and can work even if the application running on the platform is not known *a priori*. However the degradation estimations may not be as accurate due to the usage of an analytical model rather than more detailed simulation models. In the MADNESS project, we have been investigating both approaches. However, in this case, we adopt the online heuristics approach, in particular, the NMS-A/B/C heuristics proposed in [14].

To examplify, the NMS-A heuristic can be summarized as follows: let L_j be the set of tasks assigned to core n_j. L_f is the set of tasks to be migrated from the faulty node n_f. T_j^N is the sum of the execution times of tasks assigned to node n_j. $T_{cap_{ij}}^{TN}$ is the execution time of task t_i if assigned to node n_j. The task $t_i \in L_f$ is remapped on the core that minimizes its finishing time. Inputs to the NMS-A algorithm are the initial mapping L, faulty node set n_f, and T^N before the fault occurrence. The output is the new mapping L. All of the NMS-A/B/C heuristics are implemented on the MADNESS platform as a part of the runtime manager shown in Fig. 5.1, and a selected one is called upon the reception of the fault detection message from the TMH.

6.2 Experiments and results

In this section we describe a set of experiments that we performed in order to evaluate the implemented system adaptivity and fault-tolerance techniques. The application case studies are described in Section 6.2.1. We map these applications onto a 2x2 mesh of general-purpose processors, as detailed in Chapter 4, implemented on an Virtex-6 FPGA board.

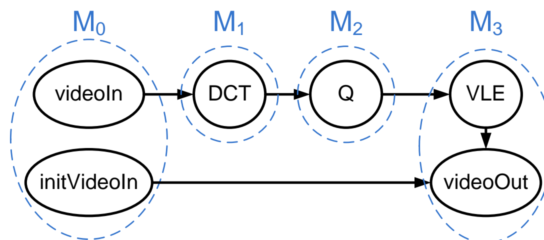

Figure 6.2: PPN specification of the M-JPEG encoder.

Table 6.1: Execution times of M-JPEG processes

Process	Avg execution time (c.c.)
M_0	1923
M_1	123626
M_2	69254
M_3	47989

Firstly, we verify that the PPN communication API enables inter-tile communication according to the PPN semantics and we compare the *passive* and *active* implementation of the middleware. Then, we present a remapping process, exploiting the migration mechanism detailed in Section 5.2. according to the on-line remapping strategies. Finally, we test the accuracy of such strategies to verify the optimality of the chosen migration decision.

6.2.1 Case studies

We chose as case studies two streaming applications in the multimedia domain, M-JPEG encoder and H.264 decoder. The two applications are described below.

M-JPEG encoder

The PPN specification of the M-JPEG encoder is shown in Fig. 6.2. The size of tokens ranges between 16 and 1024 bytes, and all of the channels are written 128 times, except the output of *initVideoIn* which is written only once per frame. Fig. 6.2 also shows how some processes have been merged to map the application on the NoC platform, e.g. *VLE* and *videoOut* processes have been merged into process M_3. The numbers of clock cycles required for the execution of each process of the M-JPEG application are summarized in Table 6.1. Comparing these numbers with the amount of inter-process communication one can infer that this application has a high computation/communication ratio.

H.264 decoder

The simplified PPN specification of this case study is shown in Fig. 6.3. In the final implementation, the nodes get_data, $parser$, and $cavlc$ have been merged into a single process, H_0. In this case study, the size of the exchanged tokens ranges between 1 and 5000 bytes. The execution time of each process of the H.264 decoder application are shown in Table 6.2.

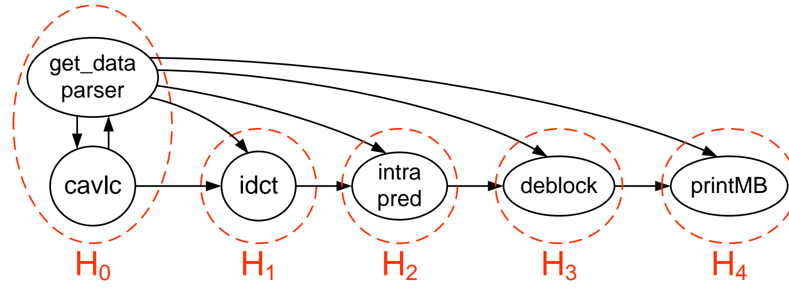

Figure 6.3: Simplified PPN specification of the H.264 decoder.

Table 6.2: Execution times of H.264 processes

Process	Avg execution time (c.c.)
H_0	95643
H_1	55775
H_2	33645
H_3	9724
H_4	4075

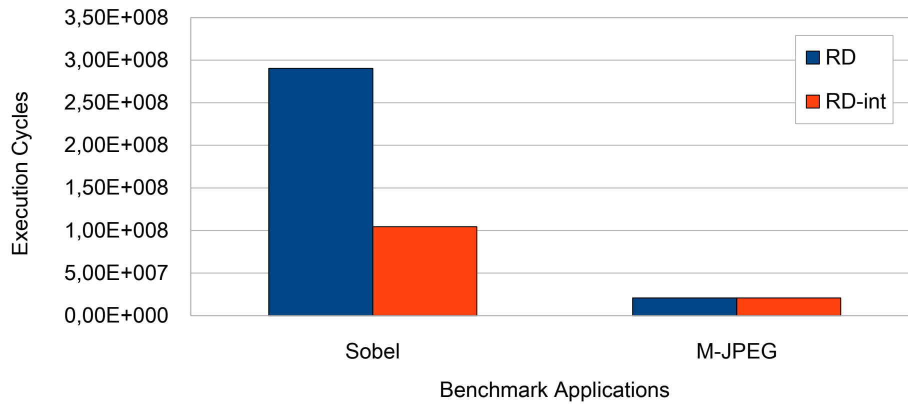

Figure 6.4: Impact of the interrupt-based request messages on the *Request-driven* flow control on two benchmark applications.

6.2.2 Flow control functionality assessment

Mapping the application on the hardware platform allowed us to test the functionality of the PPN communication APIs. We show the results obtained for the M-JPEG application. As mentioned earlier, the M-JPEG encoder is computation-intensive, so communication latencies due to the flow control do not have a deep impact on the overall performances [8]. We tested the *Request-driven* flow control by comparing the previously proposed approach with interrupt-based implementation. The two approaches did not lead to significant differences in the M-JPEG case study, as shown in Fig. 6.4. Thus, in order to compare them over a more communication-intensive benchmark, we repeated the experiment executing a Sobel filtering kernel on the platform. In this case, the execution time was significantly reduced (ca. 64%), as expected.

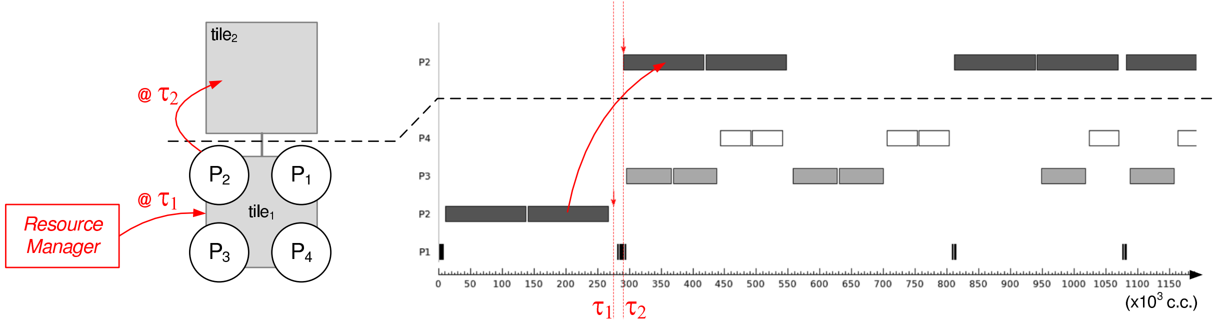

Figure 6.5: M-JPEG process scheduling when migrating M_1 using the proposed remapping heuristic and migration mechanism.

6.2.3 Remapping heuristic and process migration execution time overhead

We evaluate the proposed process migration mechanism and remapping heuristic overhead using the setup shown in the left part of Fig. 6.5. Processes $M_0 - M_3$ in the figure refer to the specification represented in Fig. 6.2. Initially, M_0 is mapped on $tile_3$, M_1 on $tile_1$, M_2 and M_3 on $tile_4$. This process mapping results in a total execution time of the M-JPEG application of $T_{exe}(noMig) = 17,332,807$ clock cycles (c.c.) in case of no migration.

However, in this experiment at time τ_0 we trigger an interrupt on $tile_3$, which activates the *run-time manager*. This interrupt emulates a message sent from the TMH of $tile_1$, indicating that the processing element is faulty. Thus, the processes running on it have to be migrated on other tiles. The migration procedure is then started. It can be divided in the timing intervals shown in the right part of Fig. 6.5 and described below.

- **$[\tau_0, \tau_1]$:** this is the time required by the *run-time manager* to make the remapping decision
- **$[\tau_1, \tau_2]$:** in this time interval the *source tile* ($tile_1$) sends all the process state to the *destination tile*
- **$[\tau_2, \tau_3]$:** between these two instants the *destination tile* ($tile_2$) copies the process state to its local memory and starts the execution of the migrated process

In total, the migration procedure takes $(\tau_3 - \tau_0) = 28,934$ clock cycles. Note that the execution of the migrated process has to be restarted from the beginning of the interrupted iteration. Thus, all the time spent since the beginning of the interrupted iteration on $tile_1$ has to be added to the total overhead. The worst case overhead due to the re-execution of the interrupted iteration is as large as the execution time of a whole iteration of the interrupted process, in this case M_1. The worst-case total overhead in the scenario depicted in Fig. 6.5 then grows up to $(\tau_3 - \tau_0) + T_{exe}(M_1) = 152,560$ clock cycles. Compared to the total execution time of the application without migration ($T_{exe}(noMig)$), this represents only 0.88% of the time.

To evaluate the calculation time of the remapping decision, the two remapping scenarios given in Fig. 6.6 and 6.9 are used for M-JPEG and H.264 applications. The NMS-A/B/C heuristics from [14], which aim at minimizing the throughput degradation, are implemented

Table 6.3: Calculation times of remapping heuristics

Heuristic	Avg execution time (c.c.)	
	M-JPEG	H.264
NMS-A	8198	8172
NMS-B	19608	19603
NMS-C	6403	6664

on the platform. Their calculation time are displayed in Table 6.3. The results reveal that their execution time constitutes a relatively small portion of the migration overhead.

6.2.4 Evaluation of the remapping strategy

In this section, the quality of the heuristic is evaluated using the M-JPEG and H.264 case studies by comparing the remapping obtained by the NMS-A/B/C heuristics with actual measurements.

M-JPEG remappings

Given a 2x2 NoC-based platform with processing elements ($tile_1 = n_1, tile_2 = n_2, tile_3 = n_3, tile_4 = n_4$) and an initial mapping of M-JPEG tasks $I : M_0 \to n_3$, $M_1 \to n_1$, $M_2 \to n_2$, $M_3 \to n_4$ as shown in Fig. 6.6(a), we consider two single fault scenarios for n_1 and n_2. As shown in Fig. 6.6(b), for the case of n_1 faulty, all possible remappings are R_1 ($M_1 \to n_2$), R_2 ($M_1 \to n_3$) and R_3 ($M_1 \to n_4$). Similarly, Fig. 6.6(c) shows the case of n_2 faulty for which all possible remappings are R_1 ($M_2 \to n_1$), R_2 ($M_2 \to n_3$) and R_3 ($M_2 \to n_4$). The total execution times of the M-JPEG application for all possible remappings, T_{R_i}, are measured on the platform using the RD-int flow control and also calculated by the analytical model.

The performance degradation with respect to the execution time of the initial mapping, T_I, is calculated according to Equation 6.1.

$$Performance\ degradation(R_i) = \frac{T_{R_i} - T_I}{T_I} \tag{6.1}$$

Measured and calculated values are used in Equation 6.1 for calculating the *measured* and *analytical model* degradation results shown in Fig. 6.7 and 6.8 for faulty n_1 and faulty n_2 cases, respectively. Note that in some cases, for instance *R2* in Fig. 6.7, the remapping can lead to a performance speedup. In *R2*, this is because the reduction of the communication time over the NoC overcompensates the increased computational workload on n_3.

The optimal remapping is the one which yields to the smallest performance degradation. For the faulty n_1 scenario, all of the NMS-A/B/C heuristics yield to the remapping R_2 which is the optimal decision. For the faulty n_2 scenario, it yields to the remapping R_2 which is only .07% worse than the optimal one (R_3). NMS-A/B/C heuristics make the optimal decision according to the analytical model and the discrepancy between the analytical model and the actual measurements causes a very slightly sub-optimal decision in reality. However, as shown in Fig. 6.7 and 6.8, the analytical model estimates the degradation within 3% of the measured values. The inaccuracy of the analytical model is due to the blockings in the communication and the unaccounted context switching times when several tasks are running on a processor.

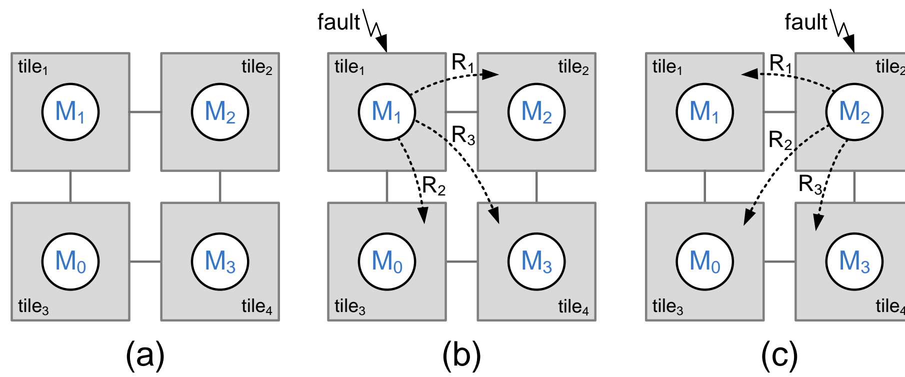

Figure 6.6: Initial mapping and the two single fault scenarios showing all possible remappings.

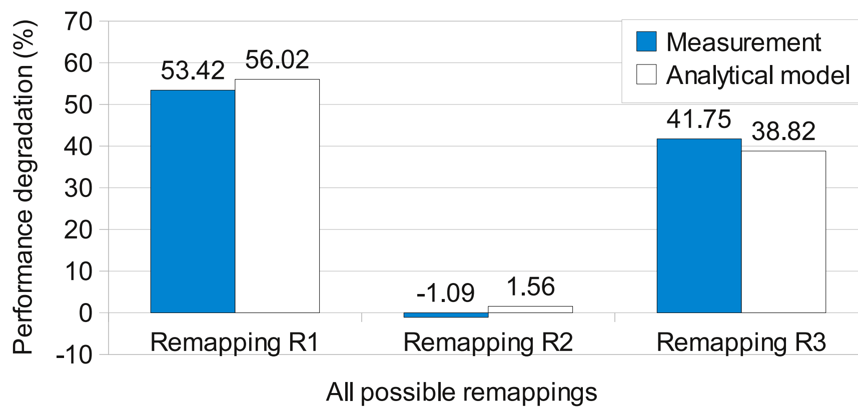

Figure 6.7: Comparison of measured and calculated performance degradation of all possible remappings when n_1 is faulty as shown in Fig. 6.6(b).

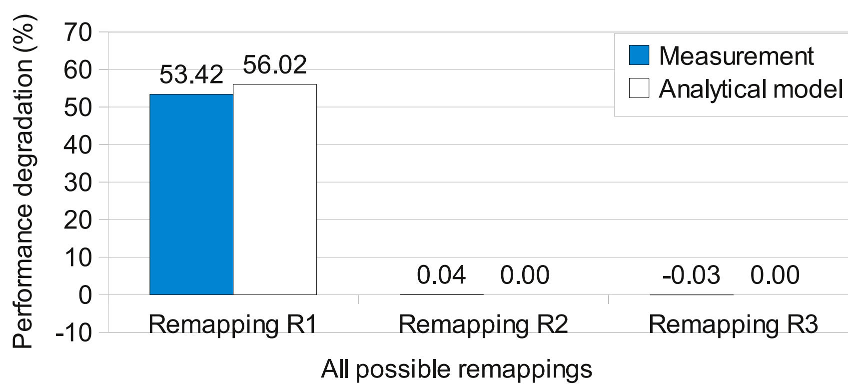

Figure 6.8: Comparison of measured and calculated performance degradation of all possible remappings when n_2 is faulty as shown in Fig. 6.6(c).

H.264 remappings

We use the same procedure to assess the NMS-A/B/C remapping heuristics in the H.264 case study. The initial mapping is shown in Fig. 6.9(a). Then, we consider the case of a fault

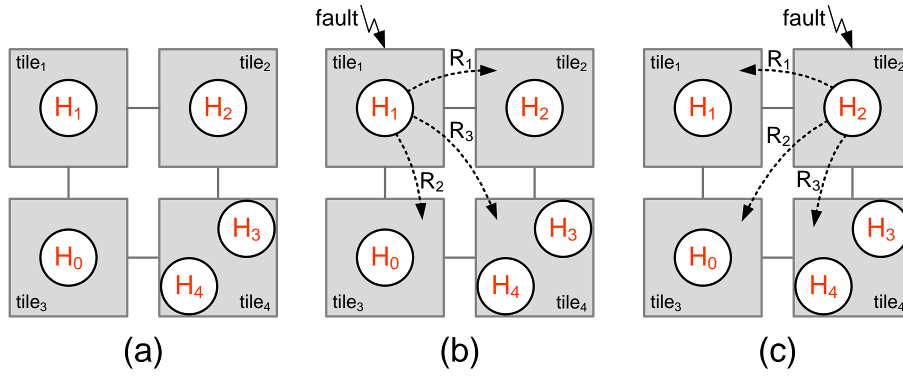

Figure 6.9: Initial mapping and the two single fault scenarios showing all possible remappings.

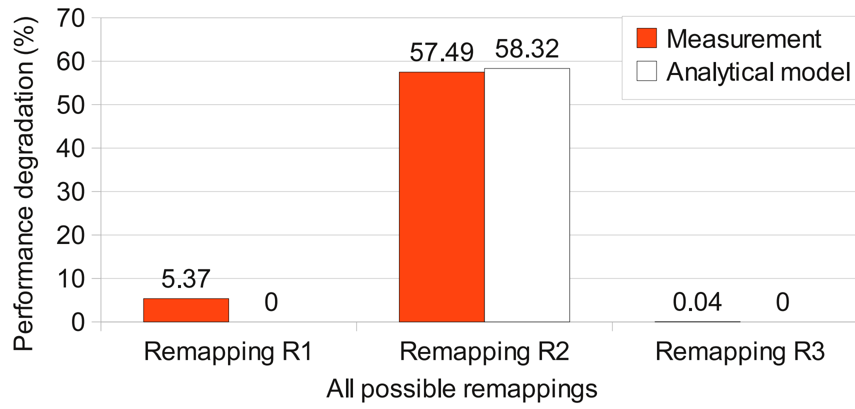

Figure 6.10: Comparison of measured and calculated performance degradation of all possible remappings when n_1 is faulty as shown in Fig. 6.9(b).

occurring either in n_1 or n_2. In each of these cases there are three possible remappings (R_1 to R_3), which are depicted in Fig. 6.9(b) and Fig. 6.9(c).

In case of a fault occurring on n_1, all of the NMS-A/B/C heuristics yield to remapping R_3, which is the optimal one as shown in Fig. 6.10. In the other considered case, faulty n_2, all the heuristics suggest remapping R_3. Also in this case, the suggested remapping represents the optimal one, as can be deduced by Fig. 6.11. Similar to the M-JPEG experiments, the inaccuracy of the analytical model is due to the abstraction of the overheads related to context switches and communication over the platform.

6.2.5 Architectural support hardware overhead

Obviously, the circuitry implementing the support for adaptivity and fault-tolerance at architectural level incurs an overhead in terms of area obstruction, power consumption and critical path length. To evaluate the overhead, we consider the basic ×pipes mesh as a baseline architecture. As mentioned earlier, with respect to the baseline, the Network Adapter has been enriched with the DMA message-passing handler (MPH). It provides all the message passing capabilities that are needed to implement the inter-processor communication, the triggering of the migration process and the migration process itself. Moreover, this mod-

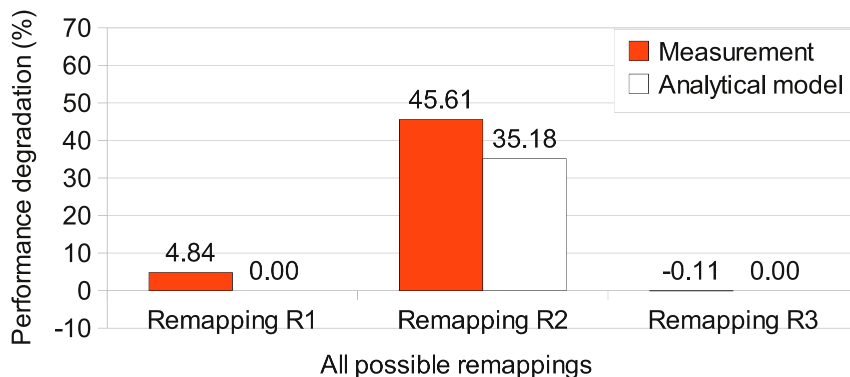

Figure 6.11: Comparison of measured and calculated performance degradation of all possible remappings when n_2 is faulty as shown in Fig. 6.9(c).

ule allows the possibility of intra-processor multitasking. Controlling the local memory, to store the incoming messages when a receive() has not been performed, the MPH allows, at the producer side, scheduling a different task when waiting for requested tokens, without stalling on a blocking receive primitive. Thus, the MPH can be considered as a first level of architectural support for adaptivity. The second level is represented by the insertion of the TMH, that has to take care of sending the migration data of the processes in case of faulty processing element. In Figure 6.12 and Figure 6.13, a preliminary estimation of the overhead due to the introduction of these modules is shown in terms of area occupation and maximum working frequency, respectively. The implementation results are obtained by means of the Xilinx tools during the protoyping phase.

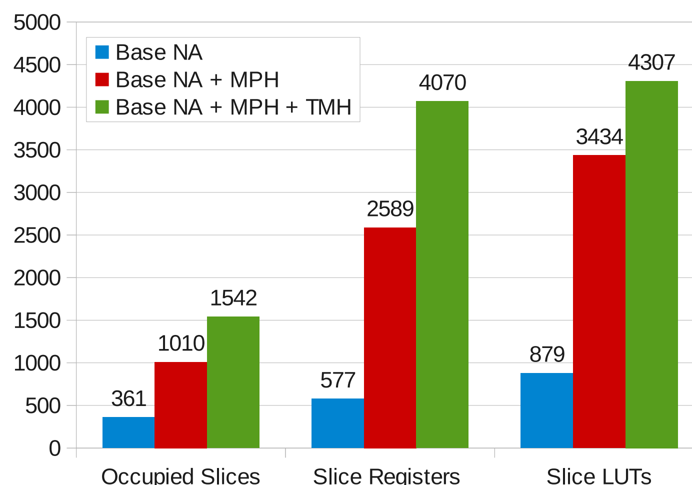

Figure 6.12: Area occupation overhead in comparison to the baseline network adapter due to the support for system adaptivity and fault-tolerance

It can be noticed that the overhead is not negligible. In terms of timing, the baseline architecture can be more than 25% faster than the NA featuring full support for fault-tolerance,

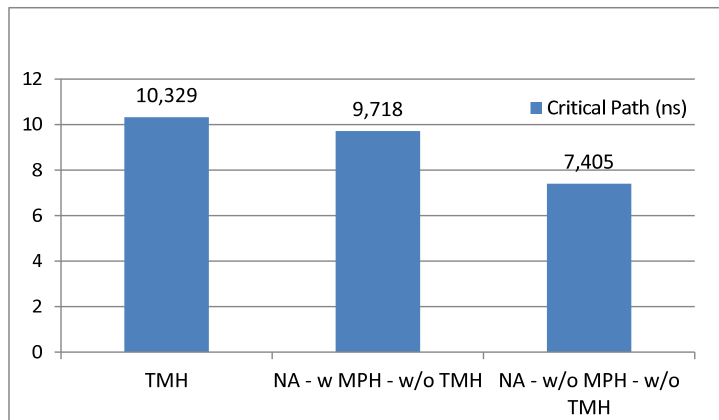

Figure 6.13: Critical path length overhead related with support for system adaptivity and fault-tolerance

especially due to the introduction of the the MPH. During the design of the MPH architecture we tried to reduce as much as possible the latency related with message passing operations. This required the introduction of combinational logics which resulted in the mentioned frequency drop. A retiming of the control circuitry inside the MPH could be used to improve the achievable working frequency, at the price of an increment of the communication latency for each packet. The overhead in terms of used logic is also significant. Such overhead is mitigated when we consider the area of the entire tile, as shown in Figure 6.14. In this case the area overhead in a tile with full support for adaptivity and fault-tolerance is almost 60% with respect to a tile instantiating the baseline NA. This overhead would be even smaller if we consider in the baseline area all the obstruction related to the memory modules, not accounted in the presented plot. It is also worth to notice that the baseline architecture cannot provide complete message-passing capabilities, thus is not completely sufficient even in static message-passing systems.

Moreover, it is useful to point out that both the MPH and the TMH can be customized at design time, according to the communication graph of the target application, instantiating only the circuitry needed to control the required number of channels and tasks. As an example, we show how the TMH is customized for the H264 and the MJPEG applications. In the first design case, the TMH has to support 4 tasks and 4 channels, requiring 35 registers to be instantiated. In the second, the circuitry must control 5 tasks and 8 channels, requiring 51 registers. The overhead comparison with the default configuration is shown in Figure 6.15.

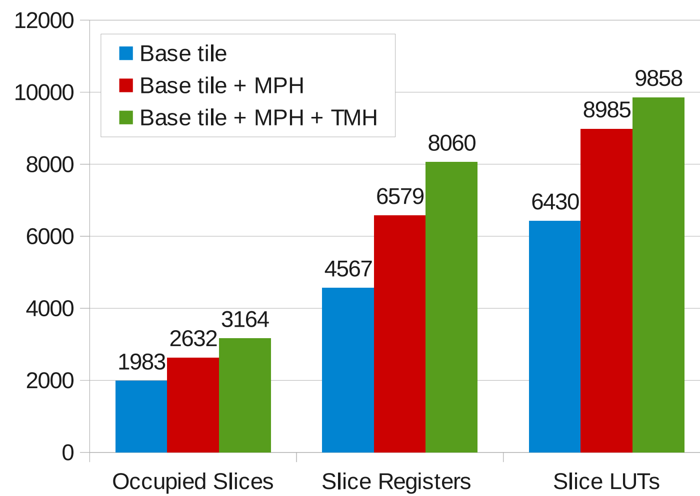

Figure 6.14: Area occupation overhead in comparison to the baseline tile architecture due to the support for system adaptivity and fault-tolerance

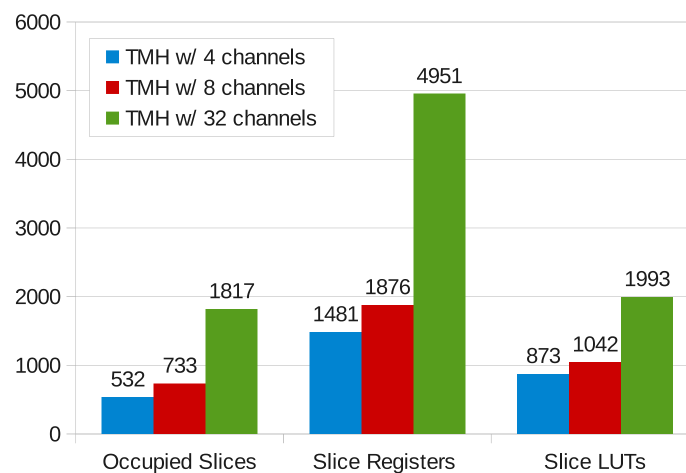

Figure 6.15: Area overhead dependence on the supported number of channels

Chapter 7

Conclusions and future developments

In this thesis, we addressed the problem of achieving system adaptivity on generic tiled NoC-based MPSoC platforms, by proposing an approach that enables the run-time migration of processes among the available platform resources. We also described how the use of reconfiguration strategies can be exploited when detecting run-time faults, to cope with the problem of graceful degradation of the system.
At first, in Chapter 2, we provided an in-depth survey concerning the state-of-the-art in system adaptivity and fault-tolerance, highlighting the challenges posed by modern embedded systems, that often need a guaranteed degree of fault-tolerance. Then, in Chapter 3, we presented a comprehensive overview of the MADNESS project, ranging from the scenario-based DSE methodologies, to the introduction of the strategies that take profit from the flexibility of the NoC-based platform. The described techniques allow a DSE driven by the requirement of different workloads, and also support the coexistence of different mapping configurations. In Chapter 4, we described the FPGA-based MADNESS evaluation platform for the exploration and characterization of adaptive and fault-tolerant MP-SoC architectures. We have been particularly focused on the hardware extensions added in order to support the PPN MoC. We also developed a predictable and efficient process migration mechanism, that should allow the system to survive in case of permanent faults, as described in detail in Chapter 5. We introduced a middleware support for the execution of Polyhedral Process Networks on Network-on-Chip MPSoCs allowing system adaptivity. Two main middleware components have been devised and implemented.

The first one is the *PPN communication API*, which realizes the PPN semantics on NoC implementations. Three communication approaches have been evaluated experimentally on two applications with very different computation and communication characteristics. The results show that the *virtual connector* approach outperforms the others when implementing communication-dominant applications. However, especially for this kind of applications, the price we pay for system adaptivity and generality is large in terms of performance, if compared to customized point-to-point systems. On the contrary, when the computation/communication ratio of an application is higher, the overhead introduced by the execution on NoC with all the proposed communication approaches is much lower.

The second middleware component concerns the process migration procedure, which

is essential for system adaptivity. A reactive and predictable process migration mechanism has been devised and developed. The proposed mechanism does not need user-specified checkpointing since it exploits the simple structure of PPN processes, whose state is only represented by iterator sets and the content of input/output FIFO buffers. Moreover, it allows the execution of a migration at any time during the execution of the main body of processes, since it does not rely on fixed migration points. The proposed migration mechanism is predictable because an upper bound of its overhead can be derived, for GT NoCs, from the process network topology and buffer sizes. Moreover, we show that the migration mechanism allows the system to change its performance metrics at run-time with almost negligible overhead.

Finally, in Chapter 6 we described how the presented process migration mechanism can be exploited by the run-time manager to cope with permanent faults by migrating the processes running on the faulty processing element. A fast heuristic is used to determine the new mapping of processes to tiles. We also proved, with a real-life case study, that this heuristic is able to find near-optimal remappings. Moreover, the experimental results prove that the overhead in terms of execution time due to the execution of the remapping heuristic, together with the actual process migration, is almost negligible compared to the execution time of the whole application. This means that the MADNESS fault-tolerance approach allows the system to react to faults without a substantial impact on the user experience. On the other hand, the support at architecture level has a significant overhead that would have to be carefully assessed and limited.

To conclude, we can say that the presented use cases, and their stable results confirm the usefulness of the developed approaches. Nevertheless, the proposed techniques have been tailored targeting application domains characterized by multimedia streaming workloads. The execution on the same platform of applications with different levels of criticality deeply impacts the complexity of the design process, in a variety of aspects, that require specific support at different levels: the hardware architecture should provide specific mechanisms to ease time-predictability and adaptation of spatial and temporal allocation of the computing resources; moreover, the software run-time should provide specific scheduling and mapping support, in order to optimize the execution of the non-critical workload part. We already planned to face the challenge of adding mixed-criticality support to the developed platform, by researching and developing novel design methods that will allow for combining these new different requirements, models and techniques into a single, automated design flow for MPSoCs.

Bibliography

[1] A high performance message passing library. [cited at p. 5]

[2] Multicore associations communication api. [cited at p. 5]

[3] Andrea Acquaviva, Andrea Alimonda, Salvatore Carta, and Michele Pittau. Assessing task migration impact on embedded soft real-time streaming multimedia applications. *EURASIP J. Emb. Sys.*, 2008, 2008. [cited at p. 4, 7, 43]

[4] Gabriel Marchesan Almeida, Gilles Sassatelli, Pascal Benoit, Nicolas Saint-Jean, Sameer Varyani, Lionel Torres, and Michel Robert. An Adaptive Message Passing MPSoC Framework. *Int. J. of Reconfigurable Computing*, 2009:20, 2009. [cited at p. 4, 6]

[5] Federico Angiolini, Paolo Meloni, Salvatore M. Carta, Luigi Raffo, and Luca Benini. A layout-aware analysis of networks-on-chip and traditional interconnects for mpsocs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(3), march 2007. [cited at p. 15]

[6] Iuliana Bacivarov, Wolfgang Haid, Kai Huang, and Lothar Thiele. Methods and Tools for Mapping Process Networks onto Multi-Processor Systems-On-Chip. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 1007—1040. Springer, October 2010. [cited at p. 5]

[7] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '06, pages 15–20, 2006. [cited at p. 4, 7, 43]

[8] Emanuele Cannella, Onur Derin, Paolo Meloni, Giuseppe Tuveri, and Todor Stefanov. Adaptivity Support for MPSoCs based on Process Migration in Polyhedral Process Networks. *VLSI Design*, 2012:17 pages, 2012. [cited at p. 60]

[9] Thomas L. Casavant, Jon, and G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14:141–154, 1988. [cited at p. 4]

[10] Chen-Ling Chou and R. Marculescu. Farm: Fault-aware resource management in noc-based multiprocessor platforms. In *Design, Automation Test in Europe Conf. Exh. (DATE), 2011*, pages 1 –6, march 2011. [cited at p. 7]

[11] Matteo Dall'Osso, Gianluca Biccari, Luca Giovannini, Davide Bertozzi, and Luca Benini. Xpipes: a Latency Insensitive Parameterized Network-on-Chip Architecture for Multi-Processor SoCs. In *Proc. of the 21st Int. Conf. on Computer Design*, ICCD'03, pages 536–, Washington, DC, USA, 2003. [cited at p. 23, 29]

[12] Giovanni De Micheli and Luca Benini. *Networks on Chips: Technology and Tools*. Morgan Kaufmann, 2006. [cited at p. 3, 15]

[13] Onur Derin, Erkan Diken, and Leandro Fiorin. A middleware approach to achieving fault-tolerance of kahn process networks on networks-on-chips. *International Journal of Reconfigurable Computing*, 2011(Article ID 295385), February 2011. [cited at p. 5, 37, 55]

[14] Onur Derin, Deniz Kabakci, and Leandro Fiorin. Online task remapping strategies for fault-tolerant network-on-chip multiprocessors. In *Proc. of the 5th ACM/IEEE Int. Sym. on Networks-on-Chip*, pages 129–136, 2011. [cited at p. 58, 61]

[15] S. V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthoor, F. Vandeputte, and K. De Bosschere. System-scenario-based design of dynamic embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 14(1):1–45, 2009. [cited at p. 12]

[16] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, and Koen De Bosschere. System-scenario-based design of dynamic embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):3:1–3:45, January 2009. [cited at p. 1]

[17] Wolfgang Haid, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Multiprocessor SoC software design flows. *IEEE Signal Processing Magazine*, 26, 2009. [cited at p. 5]

[18] Wolfgang Haid, Lars Schor, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed fifos. In *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 35–44, Grenoble, France, 2009. IEEE. [cited at p. 5, 6]

[19] Mike Holenderski, Martijn M.H.P. van den Heuvel, Reinder J. Bril, and Johan J. Lukkien. Grasp: Tracing, visualizing and measuring the behavior of real-time systems. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, July 2010. [cited at p. 50]

[20] J.Y. Hur, S. Wong, and S. Vassiliadis. Partially reconfigurable point-to-point fpga interconnects. *International Journal of Electronics*, 95:725–742, July 2008. [cited at p. 1]

[21] IEEE Computer Society and IEEE Standards Association Corporate Advisory Group. *IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows*, February 2010. [cited at p. 11]

[22] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974. [cited at p. 3, 10]

[23] Seongnam Kwon, Yongjoo Kim, Woo-Chul Jeun, Soonhoi Ha, and Yunheung Paek. A retargetable parallel-programming framework for mpsoc. *ACM Trans. Des. Autom. Electron. Syst.*, 13:39:1–39:18, July 2008. [cited at p. 5]

[24] Chanhee Lee, Hokeun Kim, Hae-woo Park, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. A task remapping technique for reliable multi-core embedded systems. In *Proc. of the 8th Int. Conf. on Hardware/software codesign and system synthesis*, pages 307–316, 2010. [cited at p. 7]

[25] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. [cited at p. 5]

[26] P. Meloni, I. Loi, F. Angiolini, S. M. Carta, M. Barbaro, L. Raffo, and L. Benini. Area and power modeling for networks-on-chip with layout awareness. *VLSI DESIGN*, 2007, 2007. [cited at p. 13]

[27] P. Meloni, S. Pomata, L. Raffo, R. Piscitelli, and A. D. Pimentel. Combining on-hardware prototyping and high-level simulation for dse of multi-asip systems. In *Proc. 12th International Conference on Embedded Computer Systems (SAMOS-XII)*, 2012. [cited at p. 13]

[28] P. Meloni, S. Secchi, and L. Raffo. An fpga-based framework for technology-aware prototyping of multicore embedded architectures. *Embedded Systems Letters, IEEE*, 2(1):5 –9, 2010. [cited at p. 13]

[29] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32:241–299, September 2000. [cited at p. 4]

[30] Dmitry Nadezhkin, Sjoerd Meijer, Todor Stefanov, and Ed Deprettere. Realizing FIFO Communication When Mapping Kahn Process Networks onto the Cell. In *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, pages 308–317, Berlin, Heidelberg, 2009. Springer-Verlag. [cited at p. 5, 6, 39]

[31] A. B. Nejad, K. Goossens, J. Walters, and B. Kienhuis. Mapping kpn models of streaming applications on a network-on-chip platform. In *ProRISC 2009: Proceedings of the Workshop on Signal Processing, Integrated Systems and Circuits*, November 2009. [cited at p. 6]

[32] André Nieuwland, Jeffrey Kang, Om Prakash Gangwal, Ramanathan Sethuraman, Natalino Busá, Kees Goossens, Rafael Peset Llopis, and Paul Lippens. C-heap: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems*, 7:233–270, 2002. [cited at p. 5]

[33] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and Automated Multiprocessor System Design, Programming, and Implementation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(3):542–555, 2008. [cited at p. 5, 10, 48]

[34] Vincent Nollet, Diederik Verkest, and Henk Corporaal. A Safari Through the MPSoC Run-Time Management Jungle. *Signal Processing Systems*, 60(2):251–268, 2010. [cited at p. 4, 15]

[35] (OCP-IP). Open Core Protocol Standard, 2003. http://www.ocpip.org/home. [cited at p. 17, 21]

[36] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. Robust optimization of soc architectures: A multi-scenario approach. In *ESTImedia*, pages 7–12, 2008. [cited at p. 1]

[37] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112, 2006. [cited at p. 10, 11]

[38] M. Psarakis, D. Gizopoulos, M. Hatzimihail, A. Paschalis, A. Raghunathan, and S. Ravi. Systematic software-based self-test for pipelined processors. In *43rd Design Automation Conf.,*, pages 393–398, 2006. [cited at p. 55]

[39] S. Secchi, P. Meloni, and L. Raffo. Exploiting FPGAs for technology-aware system-level evaluation of multi-core architectures. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 194 –202, march 2010. [cited at p. 17]

[40] Jonathan M. Smith. A survey of process migration mechanisms. *SIGOPS Oper. Syst. Rev.*, 22, July 1988. [cited at p. 4]

[41] William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 365–376, 2010. [cited at p. 5]

[42] M. Thompson, T. Stefanov, H. Nikolov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In *Proc. of the Int. Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS '07)*, pages 9–14, 2007. [cited at p. 10]

[43] P. van Stralen and A. D. Pimentel. Scenario-based design space exploration of MPSoCs. In *Proc. of the IEEE International Conference on Computer Design (ICCD '10)*, Oct. 2010. [cited at p. 12]

[44] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte. The molen polymorphic processor. *IEEE Trans. Comput.*, 53(11):1363–1375, November 2004. [cited at p. 1]

[45] Sven Verdoolaege. *Handbook on signal processing systems*, chapter Polyhedral process networks. Springer, 2010. [cited at p. 3, 4]

[46] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. pn: a tool for improved derivation of process networks. *EURASIP J. Embedded Syst.*, 2007, January 2007. [cited at p. 4, 7, 34]

[47] Xilinx. MicroBlaze Processor Reference Guide UG081.(v 9.0), 2010. http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf. [cited at p. 22]

[48] Xilinx. PowerPc Processor Reference Guide UG011.(v 1.3), 2010. http://www.xilinx.com/support/documentation/user_guides/ug011.pdf. [cited at p. 22]

List of publications related to the thesis

Published papers

Journal papers

- E. Cannella, O. Derin, P. Meloni, G. Tuveri and T. Stefanov, *Adaptivity Support for MPSoCs based on Process Migration in Polyhedral Process Networks* VLSI Design, vol. 2012, Article ID 987209, 17 pages. (Relation to Chapter 5)

Conference papers

- P. Meloni, G. Tuveri, L. Raffo, E. Cannella, T. Stefanov, O. Derin, L. Fiorin and M. Sami, *System Adaptivity and Fault-tolerance in NoC-based MPSoCs: the MADNESS Project Approach* in 15th Euromicro Conference on Digital System Design Cesme, Turkey, September 2012 (Relation to Chapter 6)

- O. Derin, P. Kuncheerath, P. Meloni, G. Tuveri *A Low Overhead Self-adaptation Technique for KPN Applications on NoC-based MPSoCs* in 3rd International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS) - Special Session on Self-Adaptive Networked Embedded Systems (SANES) Barcelona, Spain, February 2013 (Relation to Chapter 6)

Posters with published proceedings

- G. Tuveri, S. Pomata, S. Secchi and P. Meloni *A configurable and scalable multi-core architecture template supporting hybrid model of computation* in Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2011), Fiuggi, Italy, July 2011 (Relation to Chapter 4)

List of publications unrelated to the thesis

Published papers

Journal papers

- P. Meloni, S. Pomata, G. Tuveri, S. Secchi, L. Raffo, and M. Lindwer *Enabling fast ASIP design space exploration: an FPGA-based runtime reconfigurable prototyper* VLSI Design, vol. 2012, Article ID 580584)

Conference papers

- S. Pomata, P. Meloni, G. Tuveri, L. Raffo and M. Lindwer *Enabling fast ASIP design space exploration: an FPGA-based runtime reconfigurable prototyper* in Design, Automation and Test in Europe Conference Exhibition(DATE 2012), Dresden, Germany, March 2012)

- L. Jozwiak, M. Lindwer, R. Corvino, P. Meloni, L. Micconi, J. Madsen, E. Diken, D. Gangadharan, R. Jordans, S. Pomata et al., *ASAM: Automatic Architecture Synthesis and Application Mapping* in 15th Euromicro Conference on Digital System Design, Cesme, Turkey, September 2012)

Posters with published proceedings

- S. Pomata, G. Tuveri, P. Meloni, M. Lindwer *Fast ASIP Design Space Exploration on FPGAs through Binary Translation* in Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 11, Fiuggi, Italy, July 2011)